Departamento de
Universidade de Aveiro Electrónica, Telecomunicações e Informática,
2007

Miguel Augusto Mendes Oliveira e Silva

Metodologias e Mecanismos para Linguagens de Programação Concorrente Orientadas por Objectos


Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2007

Miguel Augusto Mendes Oliveira e Silva

Metodologias e Mecanismos para Linguagens de Programação Concorrente Orientadas por Objectos

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Informática, realizada sob a orientação científica de José Alberto Rafael, Professor do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho à Paula, à Ana Miguel e ao João José.

o júri / the jury

presidente / president

José Joaquim Cristino Teixeira Dias
Professor Catedrático da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)

vogais / examiners committee

José Alberto dos Santos Rafael
Professor Associado da Universidade de Aveiro (orientador)

Pedro João Valente Dias Guerreiro
Professor Associado da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Pedro Manuel Rangel Santos Henriques
Professor Associado da Escola de Engenharia da Universidade do Minho

António Manuel de Brito Ferrari de Almeida
Professor Catedrático da Universidade de Aveiro

António Rui Oliveira e Silva Borges
Professor Associado da Universidade de Aveiro

agradecimentos / acknowledgements

Os meus mais profundos agradecimentos vão, em primeiro lugar, para a minha família, por estarem sempre do meu lado. Aos meus (muitos) amigos sem os quais a vida seria uma chatice. Aos meus colegas pela ajuda que nunca me negligenciaram. Ao meu orientador pela paciência e compreensão que sempre mostrou ter pelos meus atrasos crónicos (e vergonhosos) no processo de escrita desta tese. Ao Tomás pela ajuda na revisão da tese e pelo apoio que sempre me deu. Por fim, um agradecimento muito especial ao João Rodrigues, sem o qual esta tese nunca teria chegado onde chegou. A paciência, espírito critico e interesse que sempre mostrou pelo meu trabalho foram uma ajuda insubstituível.

Resumo

Esta tese faz uma aproximação sistemática à integração de mecanismos de programação concorrente em linguagens orientadas por objectos com suporte à programação por contrato e sistema de tipos estático. Nessa integração deu-se prioridade à expressividade, segurança, abstracção e realizabilidade dos mecanismos propostos. É sustentado que essa integração deve possuir ambos os modelos de comunicação entre processadores – por mensagens e partilha de objectos – e que a sincronização seja automática e abstracta. Todos os aspectos de sincronização de objectos – intra-objecto, condicional e inter-objecto – são contemplados e integrados de uma forma segura e sinérgica com mecanismos de linguagens sequenciais orientadas por objectos. É proposta e parcialmente desenvolvida uma linguagem protótipo – denominada MP-Eiffel – onde estes mecanismos e abstracções estão a ser validados experimentalmente.

Abstract

This thesis makes a systematic approach to the integration of concurrent programming mechanisms in Design by Contract and static type system based object-oriented languages. In this integration priority was given to the expressiveness, safety, abstraction and realizability of the proposed language mechanisms. We argue that this integration should provide both models of inter-processor communication – message passing and shared objects – and that synchronization should be automatic and abstract. All aspects of object synchronization – intra-object, conditional, and inter-object – were considered and integrated in a safe and synergic way with sequential object-oriented language mechanisms. We propose and partially develop a prototype language – named MP-Eiffel – in which these mechanisms and language abstractions are being validated.

Tese - ERRATA

Miguel Oliveira e Silva

14 Setembro 2007

Foi revista a paginação da tese, tendo-se rectificado as linhas que excediam os limites do texto (indicando ao LaTeX em que sítios se pode cortar as palavras a meio).

Foram também rectificadas as referências feitas a páginas em que apareciam dois parêntesis: Por exemplo, na página 18 aparecia: ((página 7)).

Para além dessas alterações foram feitas as seguintes correcções:

1. (pág. 0) O nome do departamento é agora: Departamento de Electrónica, Telecomunicações **e Informática**;

2. (pág. 9) Na nota de rodapé acrescentou-se a palavra "grandes":

   > Actualmente pode-se identificar quatro **grandes** metodologias (...)

3. (pág. 16) Foi acrescentada uma nota de rodapé após a palavra asserções indicando que as mesmas são predicados;

4. (pág. 16) Foi acrescentada uma nota de rodapé explicando a diferença entre a notação do terno de Hoare utilizada pelo próprio, e a utilizada na tese. Faz-se referência ao livro de David Gries ("The Science of Programming") onde essa diferença é explicada:

   > **Hoare apresenta esta fórmula com as chavetas a envolver a acção em vez de envolver as asserções:** $P\ \{A\}\ R$**. Estes dois formalismos diferem apenas do detalhe de na notação original de Hoare a pós-condição só ser aplicável caso a acção termine (correcção parcial) enquanto que a notação utilizada pressupõe e impõe a terminação (em tempo finito) da acção.Para os objectivos deste trabalho, no entanto, essa diferença não nos parece ser de todo relevante.**

5. (pág. 17) A chamada à nota de rodapé número 10, foi mudada para antes dos parêntesis curvos, passando a estar após a palavra "compreensão".

6. (pág. 17) O texto desta secção foi separado com a indicação de uma nova subsecção: "Limitações", antes do parágrafo que começa a abordar as limitações da programação estruturada:

   > A programação procedimental estruturada começa a mostrar as suas limitações (...)

7. (pág. 19) Foram apagadas as palavras "de todo" No terceiro parágrafo (não muda o sentido da frase).

(o seu comportamento dentro de cada objecto, é ~~de todo~~ similar ao das variáveis das linguagens procedimentais)

8. (pág. 25) Foi acrescentada uma nota de rodapé explicando o acrónimo ADT (uma vez que esta é a sua primeira ocorrência):

   Tipo de Dados Abstracto

9. (pág. 42) Foi corrigido o problema de não aparecerem as duas notas de rodapé (39 e 40) que aparecem referenciadas na tabela (aparecem agora por baixo da tabela). As notas são as seguintes:

   [39] **Existiu uma versão anterior de 1964, conhecida por SIMULA 1.**

   [39] **A primeira versão de Ada é de 1979, mas apenas em 1995 é que a linguagem se aproximou da orientação por objectos.**

10. (pág. 45) Erro no texto do quinto parágrafo. É indicado sistemas de partilha de tempo não preemptivo, quando devia ser preemptivo.

    (por exemplo, em sistemas operativos de partilha de tempo ~~não~~ preemptivo (...))

11. (pág. 48) Pequena rectificação no texto do terceiro parágrafo:

    (que podem reutilizar ~~um~~ **o próprio** mecanismo de excepções da linguagem)

12. (pág. 62) Palavra corrigida. Onde está "presentação" no primeiro parágrafo passou a estar "apresentação":

    Após a **a**presentação (...)

13. (pág. 68) As figuras 5.4 e 5.5 estavam um pouco sobrepostas. O problema foi resolvido mudando uma das figuras de página.

14. (pág. 105) Palavra errada. Onde estava *cashing* passou a estar *caching*.

15. (pág. 106) Foram acrescentadas duas tabelas sintetizando algumas das interferências inseguras e sinérgicas, tratadas durante o capítulo (as tabelas estão anexadas à errata).

16. (pág. 122) Palavra errada. Onde estava "sinergico" passou a estar "sinérgico".

17. (pág. 127) Palavra a mais (quarto parágrafo):

    Por exemplo, o código apresentado na figura B.1 ~~seguir~~ – embora (...)

18. (pág. 128) Faltava a legenda da figura B.1

    Programa errado.

19. (pág. 132) As figuras B.2 e B.3 estavam um pouco sobrepostas. O problema foi resolvido mudando uma das figuras de página.

20. (pág. 132) O glossário foi aumentado com mais alguns termos (e foram corrigido a ausência de definição do termo "Escalonamento"; e a duplicação da entrada "Rotina"). O gloss rio revisto foi anexado a esta errata

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Estudada desde há mais de 40 anos nas ciências de computação, a programação concorrente, por várias razões, tem sido em grande medida ignorada e muito pouco utilizada na prática desde então. A razão principal para esta situação deve-se provavelmente à evolução exponencial – sem paralelo em nenhuma outra área da engenharia – da electrónica e da engenharia dos computadores bem retratada na conhecida previsão de Moore [Moore 65] de que em cada ano se duplicaria o número de transístores por circuito integrado[1]. Assim o desempenho dos computadores, e por arrastamento dos programas que neles são executados, tem aumentado a um ritmo elevado, relegando para segundo plano (com a excepção dos sistemas operativos) as possibilidades de aumento de desempenho abertas pela programação concorrente.

Recentemente as unidades de processamento central têm evoluído para arquitecturas paralelas (com destaque para as arquitecturas *SMP*: *Symmetric MultiProcessing* e *NUMA*: *Non-Uniform Memory Access*), o que inevitavelmente fará aumentar imenso o interesse em linguagens e metodologias de programação concorrente.

Por outro lado, a programação orientada por objectos tem vindo a estabelecer-se como uma das mais importantes metodologias na construção de programas. As vantagens relativas que lhe podemos associar são a sua adequação e flexibilidade na modelação de diferentes tipos de problemas; as suas propriedades de modularidade, reutilização e extensibilidade; e, finalmente, a sua adequação à programação por contrato e por conseguinte, a construção de programas com correcção e robustez.

Esta dissertação estuda o problema da integração de mecanismos e abstracções de programação concorrente em linguagens orientadas por objectos. A abordagem sistemática seguida privilegiou quatro aspectos:

- **Expressividade**: os mecanismos de concorrência devem abranger, com clareza e simplicidade, todas as abstracções de programação desejadas;
- **Segurança**: a segurança no uso de mecanismos de concorrência deve ser garantida, tanto quanto possível, antes do tempo de execução dos programas;
- **Abstracção**: a semântica desses mecanismos deve-se cingir às suas propriedade essenciais, evitando um acoplamento excessivo com uma qualquer realização prática;

[1] Previsão que se tem verificado com grande aproximação na prática, nos últimos 40 anos.

- **Realizabilidade**: os mecanismos devem ser tratáveis pelo sistema de compilação.

Dos resultados obtidos neste trabalho destacamos a sincronização abstracta e automática de objectos concorrentes, assim como a segurança estática e a expressividade na integração da maioria dos requisitos de programação concorrente em linguagens orientadas por objectos com suporte para a programação por contrato.

1.1 Organização

Esta tese está organizada da seguinte forma.

No capítulo 2 apresenta-se e discute-se o problema da avaliação da qualidade de linguagens de programação. Nesse sentido são apresentadas métricas e critérios de qualidade que servirão de base não só para a escolha dos mecanismos, como também servirão como guias sobre o caminho a seguir (ou não) durante o processo de construção da linguagem.

No capítulo 3 faz-se uma apresentação detalhada sobre linguagens e programação orientada por objectos sequencial. É dado um ênfase especial aos mecanismos e propriedades consideradas essenciais nessas linguagens. Serão esses mecanismos e essas propriedades que ditarão as restrições e constrangimentos a ter em conta na integração de mecanismos concorrentes, já que se pretende que essa integração não coloque minimamente em causa as qualidades da programação por objectos.

O capítulo 4 analisa as características da programação concorrente, identificando as abstracções a serem consideradas na sua integração em linguagens sequenciais.

O capítulo 5 estuda com detalhe várias aproximações à integração de mecanismos concorrentes em linguagens orientadas por objectos, tendo em consideração os vários aspectos tratados nos capítulos anteriores: os critérios de qualidade de linguagens do capítulo 2; os mecanismos e propriedades essenciais das linguagens orientados por objectos do capítulo 3; e por fim as abstracções concorrentes a ter em conta do capítulo 4. Procura-se identificar não só as aproximações que faz sentido seguir, como também aquelas que não devem ser seguidas, sendo apresentadas razões, que se espera claras, para justificar essas conclusões.

No capítulo 6 é proposta uma linguagem orientada por objectos concorrente, denominada MP-Eiffel, onde são concretizados os mecanismos discutidos no capítulo anterior. Esta linguagem é utilizada como caso de estudo da programação orientada por objectos concorrente. Deve ser referido que a implementação actual do sistema de compilação para esta linguagem ainda não está completa, pelo que não é garantida ainda total segurança estática.

Alguns aspectos, considerados importantes, relacionados com a implementação do sistema de compilação do MP-Eiffel, são apresentados em anexo.

As conclusões deste trabalho são apresentadas no capítulo 7, onde também se enumeram as contribuições feitas.

No fim desta tese (apêndice E.4) existe um glossário com a definição de muitos dos termos e das expressões utilizadas neste trabalho.

Capítulo 2

Linguagens de Programação: Critérios de Qualidade

A programação tem por objectivo encontrar soluções[1] computáveis para resolver problemas. Existindo, em geral, inúmeras soluções computáveis para os mesmos problemas, elas distinguem-se entre si por terem diferentes qualidades. Essa qualidades dependem geralmente não só do processo de construção de programas — metodologia — utilizado, como também da linguagem (ou linguagens) utilizadas para o implementar.

Neste capítulo estamos interessados em definir critérios de qualidade na avaliação e construção de linguagens de programação que potenciem o melhoramento dos vários factores de qualidade de programas, especialmente daqueles que forem mais importantes no contexto do problema a ser resolvido. Com esse objectivo serão sumariamente descritos os factores de qualidade de programas mais importantes, após o que serão apresentados os critérios de qualidade de linguagens. Serão apresentadas justificações para os critérios apresentados mostrando em que sentido eles podem melhorar os factores de qualidade de programas.

2.1 Avaliando a qualidade de programas

Os factores de qualidade de programas podem dividir-se em dois grupos [Meyer 88a, Ghezzi 91]: factores externos e factores internos. Os factores externos expressam as qualidades visíveis para os utilizadores externos de programas. Destas temos, por exemplo, a fiabilidade, a facilidade de utilização e o desempenho. Os factores internos referem-se às qualidades visíveis apenas para os programadores, tais como, por exemplo, a modularidade e a legibilidade.

É evidente que em relação ao produto final, só irão interessar as suas qualidades externas. Pouco importa se um pacote de *software* de defesa militar é modular e de fácil compreensão se um erro na entrada acciona um míssil. Apesar desta constatação, a chave para se obterem boas qualidades externas reside precisamente na qualidade dos factores internos [Meyer 88a, página 4].

[1] Genericamente designadas por *software*.

2.1.1 Correcção

Correcção é a capacidade do *software* efectuar as suas funções exactamente como definido nas suas especificações.

Este é de longe o mais importante de todos os factores de qualidade. O primeiro objectivo de um qualquer produto de *software* é resolver o problema para que foi feito. Se isso não acontece tudo o resto pouco importa.

Como decorre da definição, a correcção de um produto de *software* depende fortemente de uma especificação suficientemente precisa do comportamento que se pretende que ele tenha. Isso raramente acontece, havendo muitas vezes somente uma especificação informal usando a linguagem natural, o que favorece ambiguidades e incorrecções.

Outros dois problemas relacionados com a especificação de programas decorrem ou da especificação incompleta (sub-especificação), ou da especificação excessiva (sobre-especificação) do problema. Por um lado um problema sub-especificado, mesmo que com rigor, pode dar origem — tendo em consideração a definição dada — a um programa formalmente correcto que não resolve o problema. A sobre-especificação, por outro lado, pode excluir soluções válidas (e eventualmente melhores) para o problema, para além de afectar negativamente outros factores de qualidade como a extensibilidade.

A arte da especificação de produtos de *software* passa assim por evitar sub-especificações sem cair na tentação de sobre-especificações.

Na construção de programas, de um ponto de vista metodológico, é preferível apesar de tudo partir de especificações incompletas — já que estas podem ir sendo completadas sem o risco de um impacto excessivo nas restantes partes do programa – do que a partir de especificações excessivas.

2.1.2 Robustez

Robustez é a capacidade dos sistemas de *software* funcionarem mesmo em situações anormais.

O conceito de robustez parece ser um pouco menos claro do que o da correcção. Que sentido fará dizer que um programa é robusto se funcionar em situações imprevistas que não fazem parte da sua especificação?

Se essas situações forem parte da especificação do programa, então o problema passaria a ser de correcção e não de robustez. Assim, o papel da robustez é de alguma forma garantir que, caso aconteça alguma situação anómala, o programa termine graciosamente (sem gerar eventos catastróficos), ou de alguma forma recupere para um estado de funcionamento normal (ou seja, dentro da especificação do programa).

Num mundo ideal, onde fosse possível desenvolver programas garantidamente correctos, não haveria lugar para a robustez. No entanto, a programação existe no mundo real onde a demonstração formal da correcção de programas se restringe a um reduzido número de problemas de baixa complexidade. Por outro lado a experimentação de programas (teste em tempo de execução) mostra ter, nesse aspecto, ainda mais limitações. Citando Dijkstra [Dijkstra 72, página 6]:

> O teste de programas pode ser utilizado para mostrar a presença de erros, mas nunca para mostrar a sua ausência.[2]

Assim, na prática um programa está sujeito a erros de programação e a falhas por vezes de difícil previsibilidade e de baixa probabilidade (propriedades que quando conjugadas podem reduzir drasticamente a qualidade do *software*); como por exemplo o esgotamento da memória livre do computador ou do espaço em disco. Passar todas estas situações excepcionais para a especificação normal de um programa – prevendo, por exemplo, a ocorrência de falta de espaço em disco sempre que nele se escreve qualquer coisa – convertendo desta forma o problema da robustez num de correcção, poderá tornar a especificação do problema bastante mais complexa, degradando outros factores de qualidade como a fiabilidade e a produtividade. Todas estas razões justificam a importância deste factor de qualidade.

2.1.3 Fiabilidade

Fiabilidade é a capacidade de um sistema de *software* ser correcto e robusto.

Este factor congrega os dois anteriores, expressando no geral o grau de confiança que se pode ter num produto de *software*.

2.1.4 Extensibilidade

Extensibilidade expressa a facilidade com que produtos de *software* se adaptam a mudanças de especificações.

É outro factor importantíssimo. É muito raro um produto de *software* não sofrer durante o seu desenvolvimento ou após a sua divulgação ou comercialização, várias alterações nas suas especificações, pelo que a sua adaptabilidade a essas mudanças será uma propriedade muito desejável.

2.1.5 Reutilização

Reutilização é a capacidade de produtos de *software* serem utilizados em parte, ou na sua totalidade, para novas aplicações.

Para além das vantagens evidentes na construção de programas reutilizando tanto quanto possível componentes existentes, este factor influencia positivamente também outros factores como é o caso muito importante da correcção[3].

[2] Neste aspecto podemos estabelecer um paralelo com as ciências físicas e o critério da falsificabilidade de Karl Popper: uma teoria é científica se for falsificável. Ou seja – tal como na programação – uma teoria cientifica tem de testável por forma a verificar se é falsa (a "verdade" é aproximada dessa forma por exclusão de partes).

[3] A correcção de um produto de *software* é tanto mais garantida quanto mais utilizado ele tenha sido no passado.

2.1.6 Eficiência

> Eficiência expressa a capacidade de usar de uma forma óptima os recursos do *hardware* (CPU, memória, etc.).

Este factor é geralmente associado à rapidez, ou desempenho do *software*. Apesar de esta ser, em geral, a medida de eficiência mais importante, outras há que poderão ser também importantes, como por exemplo o uso da memória.

2.1.7 Verificabilidade

> Verificabilidade é a capacidade de facilmente elaborar procedimentos e dados de teste para detectar erros e falhas.

Dificilmente existirá algum produto de *software* minimamente complexo que não tenha tido erros ou falhas na sua concepção. Como tal, por forma a maximizar o mais possível a sua correcção, é importante que ele seja desenvolvido facilitando a elaboração de procedimentos de teste para a detecção de erros. Desenvolver software ignorando ou minorando a possibilidade de erros comprometeria seriamente a sua correcção.

2.1.8 Produtividade

> Produtividade expressa o rendimento com o que se desenvolvem produtos de *software*.

A medida mais importante de produtividade é o tempo de desenvolvimento do *software*, embora o conceito de produtividade possa ter um sentido mais amplo, como seja o da utilização de recursos humanos e logísticos (aspecto completamente fora do âmbito deste trabalho).

2.1.9 Outros factores externos

Podem ser definidos outros factores externos de qualidade:

Compatibilidade: facilidade com que produtos de *software* são combinados uns com os outros;

Facilidade de utilização: facilidade com que se utilizam programas;

Portabilidade: facilidade com que se transportam programas para diferentes contextos de execução.

Estes factores, no entanto, não têm a mesma importância para este trabalho do que os atrás definidos.

Como é evidente, em muitas situações terá de haver soluções de compromisso entre alguns destes factores. Por exemplo, maximizar o desempenho (se levado ao extremo) pode levar a uma baixa portabilidade, ou mesmo a problemas subtis de correcção.

2.1.10 Legibilidade

Este factor interno é particularmente importante.

> A legibilidade expressa a facilidade com que se apreende e se compreende a estrutura e código de produtos de *software*.

Os programas devem ser construídos por forma a serem fáceis de ler e compreender. A legibilidade de programas – muito mais do que a facilidade em os escrever [Hoare 73, página 3] – é um critério essencial para melhorar a sua correcção. No entanto, sendo o *software* em geral complexo, esta é uma qualidade difícil de garantir. A legibilidade é aproximada utilizando metodologias de programação apropriadas, podendo as linguagens de programação contribuir decisivamente para esse fim.

2.1.11 Modularidade

Outro factor de qualidade interno essencial sendo mesmo determinante para melhorar muitos dos factores de qualidade externa, é a chamada modularidade.

Não é fácil uma definição precisa de modularidade. Intuitivamente é uma forma particular da separação de interesses, em que a divisão do problema se faz por unidades (módulos) individualizadas e coerentes, com valor e significado por si só.

Meyer [Meyer 88a] propõe cinco critérios para se avaliar a modularidade nos métodos de desenvolvimento de programas:

Decomposição modular: Se ajuda a decomposição do problema em sub-problemas, de tal modo que a resolução de cada um desses sub-problemas possa ser feita separadamente.

Composição modular: Se favorece a produção de unidades de *software* que possam ser livremente combinadas umas com as outras para gerar novos programas, mesmo para problemas muito diferentes daqueles para os quais foram desenvolvidas.

Compreensão modular: Se facilita a produção de unidades de *software* que sejam facilmente compreensíveis por observadores humanos (legíveis).

Continuidade modular: Se uma variação pequena nas especificações do problema resulta em alterações em um ou poucos módulos do sistema obtido por esse método.

Protecção modular: Um método satisfaz este critério se o efeito de uma situação que ocorra durante a execução de um módulo se mantiver confinada nesse módulo, ou se propague a poucos módulos vizinhos.

2.2 Critérios de qualidade de linguagens

As linguagens de programação são a mais importante das ferramentas para o desenvolvimento do *software*, dependendo em grande medida delas muitos dos factores de

qualidade (dos quais se destaca o mais importante deles todos: a correcção). A maior ou menor facilidade no projecto e desenvolvimento de software com qualidade depende em primeira linha das qualidades das linguagens de programação utilizadas.

Apesar dessa reconhecida importância, são relativamente raras abordagens objectivas e sistemáticas ao problema da qualidade de linguagens, mesmo na apresentação de linguagens em particular. As inúmeras discussões, geralmente estéreis, sobre qual a melhor das linguagens seriam bastante mais produtivas se houvesse a preocupação de clarificar diferentes critérios de qualidade.

A importância desses critérios de qualidade não se restringe à avaliação de linguagens existentes, sendo também essenciais na concepção de novas linguagens, já que permitem orientar esse processo de criação no sentido de melhorar as qualidades pretendidas.

A elaboração dos critérios aqui apresentados baseou-se essencialmente no artigo clássico de Hoare sobre esta temática [Hoare 73] e no trabalho de Meyer sobre a linguagem EIFFEL [Meyer 92]. Alguns dos critérios são, no entanto, da responsabilidade do autor, como é o caso da sinergia.

Hoare [Hoare 73] considera que, por forma a serem ferramentas de ajuda úteis, as linguagens de programação devem assistir o programador nos três aspectos mais difíceis da programação: projecto, documentação e depuração de programas.

2.2.1 Expressividade

No projecto de programas, o primeiro desafio essencial colocado a uma linguagem de programação consiste na facilidade com que a linguagem expressa os mecanismos e abstracções relevantes para o método (ou métodos) de programação que essa linguagem pretende suportar.

A linguagem deve expressar com clareza e simplicidade todas as abstracções e mecanismos de programação que pretende suportar.

A expressividade aplicada a toda uma metodologia de programação – por exemplo a programação por objectos – medirá a plenitude com que essa metodologia é realizada pela linguagem.

2.2.2 Abstracção

Desde o aparecimento das primeiras linguagens de programação – directa e intimamente ligadas ao sistema de suporte à execução dos programas – que a tendência tem sido de um distanciamento progressivo relativamente a esse *hardware*, e um aumento da abstracção com que as soluções são expressas nas linguagens (reduzindo, desta forma, a distância das metodologias e linguagens de programação com o domínio dos problemas que se pretende programar).

Parece evidente haver toda a vantagem em separar claramente a forma como os programas são expressos e construídos, da forma como são realizados e implementados nos sistemas de suporte à sua execução. Dito de outra forma, os programas devem ser explícitos quanto ao comportamento que deles se espera e não necessariamente à forma

como esse comportamento é traduzido nas linguagens de baixo nível utilizadas pelas unidades de processamento dos computadores.

É claro que este aspecto deixa em aberto sobre qual, ou quais serão as abstracções adequadas para expressar soluções para problemas. Essa abstracções dependerão em grande medida da metodologia de programação pretendida[4].

> A semântica da linguagem deve ser expressa relativamente aos aspectos importantes dos seus mecanismos, e não aos detalhes de eventuais possíveis realizações.

2.2.3 Compreensibilidade

A documentação de programas é um dos aspectos que tende a ser menos considerado em linguagens de programação – aparte do suporte para o uso de comentários – levando a que os respectivos programas sejam difíceis de compreender, de depurar e de modificar. Hoare defende que a documentação deve ser encarada como sendo uma parte integral, não só processo de desenvolvimento de programas, mas também do próprio programa.

> A linguagem deve encorajar e facilitar a escrita de programas legíveis e auto-documentados.

Embora a facilidade na escrita e a facilidade na leitura de um programa não sejam dois objectivos antagónicos (antes pelo contrário), é importante reforçar o facto de o segundo ser muito mais importante do que o primeiro. Se tal escolha alguma vez tivesse que ser feita, em geral seria de longe preferível ter uma escrita de programas mais trabalhosa se de tal opção resultasse uma mais fácil compreensão dos mesmos.

2.2.4 Segurança

O último aspecto referido por Hoare – a depuração de programas – será o que, provavelmente, mais obriga a escolhas radicais na construção de linguagens.

No desenvolvimento de programas, a depuração tende a ser a fase mais demorada, difícil e menos motivadora para os programadores. No entanto, o que dela resultar afecta directamente o critério de qualidade mais importante de todos: a correcção; pelo que se tornam extremamente importantes todas as ajudas que a linguagem possa dar nesse sentido. Essas ajudas podem existir em basicamente duas áreas: na detecção e na localização de erros.

O subconjunto de erros relativamente aos quais as linguagens de programação mais têm a obrigação de detectar, são os que se relacionam com utilizações incorrectas dos seus próprios mecanismos e respectivas abstracções.

Nesse sentido Hoare propõe o critério de segurança.

> Uma linguagem diz-se segura se os seus mecanismos e abstracções não produzirem resultados sem sentido.

[4] Actualmente pode-se identificar quatro grandes metodologias: a programação procedimental estruturada, a programação por objectos, a programação funcional e a programação lógica.

Pierce [Pierce 02, página 6] apresenta outra definição interessante de segurança:

> Uma linguagem diz-se segura se proteger as suas próprias abstracções.

Assim, um mecanismo será seguro se a sua utilização num programa só for aceite se houver a garantia de que nenhum resultado sem sentido advirá dessa sua utilização.

A segurança pode ser garantida antes dos programas serem executados (em tempo de compilação ou estaticamente), ou testada enquanto estes são executados (em tempo de execução ou dinamicamente). Obviamente que, no que diz respeito a este critério, a primeira opção é de longe preferível, já que (descontando eventuais erros de implementação dos sistemas de compilação das linguagens) é a única que garante a inexistência de determinados erros – como é o caso importante dos erros de tipos – durante o tempo de execução dos programas.

A opção de projecto de linguagens mais importante em termos de garantir a segurança das linguagens tem a ver com o chamado sistema de tipos da linguagem[5].

2.2.5 Sinergia

Um aspecto de qualidade de linguagens pouco referido (mas com certeza facilmente reconhecido), é não só o grau de integração e coesão dos vários mecanismos da linguagem entre si, mas também quando desse facto resultam mais valias com o aparecimento de novas funcionalidades, emergentes do uso conjunto desses mecanismos. Ou seja, essa propriedade avalia a possibilidade das funcionalidades do conjunto de determinados mecanismos serem mais do que a soma individual das funcionalidades dos mecanismos envolvidos. Iremos designar este critério por sinergia.

Quando possível, os mecanismos e abstracções das linguagens devem ser construídos por forma a que, quando utilizados em conjunto, gerem novas funcionalidades desde que estas sejam consentâneas com as respectivas semânticas individuais.

Um exemplo que pode ser considerando como sendo de sinergia é a recursividade de rotinas em linguagens imperativas. A funcionalidade da recursividade emerge devido à forma como são implementados os mecanismos de invocação de rotinas e de armazenamento (numa pilha) dos valores dos argumentos e variáveis locais à rotina. É claro que – como este exemplo bem o demonstra – os efeitos sinérgicos de mecanismos raramente são casuais, mas sim o resultado de um projecto cuidado desses mecanismos da linguagem.

2.2.6 Ortogonalidade

Levando mais longe esta perspectiva de se analisar as propriedades resultantes do uso conjunto de mecanismos, temos que quando o todo tem um valor (em termos de funcionalidades) inferior à soma das partes (cada um dos mecanismos vistos isoladamente), estamos com certeza na presença de problemas de segurança; quando esse valor

[5]Descrito na secção 3.1.

for superior à soma das partes, temos qualidades sinérgicas; e quando for igual, estamos na presença de mecanismos independentes ou ortogonais.

Assim, sendo que a segurança deve ser sempre garantida, temos apenas duas opções no funcionamento conjunto de mecanismos: ou devem ser sinérgicos ou ortogonais.

Os mecanismos e abstracções da linguagem são ortogonais, se funcionarem de uma forma independente.

Um exemplo notável de ortogonalidade é o projecto das instruções procedimentais estruturadas (algumas das quais podem ser vistas na figura 3.1). Assim, dentro de uma instrução condicional ou repetitiva pode-se utilizar qualquer outra instrução, potenciando de uma forma simples, o desenvolvimento de qualquer algoritmo (computável).

Um caso particular onde a ortogonalidade pode ser bastante importante é a situação – como acontece na linguagem protótipo desenvolvida no âmbito deste trabalho – em que se pretende extender uma linguagem existente com novos mecanismos para novas funcionalidades. Nessa situação, é desejável que os novos mecanismos sejam o mais possível ortogonais com a linguagem de base, por forma a que, não só se dê mais consistência e compreensibilidade à "nova" linguagem, como também se possa reutilizar o mais possível os módulos já existentes.

2.2.7 Outros critérios

Os critérios de qualidade já apresentados serão os mais importantes na avaliação de linguagens. Existem, no entanto, outros critérios que devem também ser tidos em conta.

Realizabilidade: Um mecanismo de uma linguagem de programação será realizável se existir pelo menos uma implementação, computável no sistema de compilação, que permita a geração do código executável apropriado no sistema de suporte à execução do programa.

A realizabilidade de uma linguagem, é um critério a ter-se em conta especialmente na fase de concepção de linguagens.

Eficiência de programas: A linguagem deve permitir que o respectivo sistema de compilação (ou, se for o caso, de interpretação) tenha a possibilidade de gerar programas eficientes[6].

Apesar do vertiginoso — pode-se mesmo dizer incomparável! — aumento na capacidade de processamento (e armazenamento) dos sistema de *hardware* que suportam a execução de programas, a eficiência será sempre um objectivo que não deve ser descurado na engenharia de software e muito em particular no projecto e realização de linguagens. Por muito rápido que seja o sistema de execução de um programa, este será tanto melhor aproveitado quanto mais eficientes forem os programas.

Existe ainda outro aspecto de eficiência aplicável às linguagens de programação: a eficiência na compilação. Actualmente, e desde que a linguagem seja realizável,

[6]Este factor de qualidade está definido na página 6.

este aspecto não será muito importante, já que mesmo sistemas de compilação pouco optimizados, tendem a ter um tempo real de execução relativamente baixo (e geralmente comportável).

Extensibilidade da linguagem: Extensibilidade de linguagens de programação expressa a facilidade com que se lhe adicionam novos mecanismos.

As linguagens de programação, não sendo de forma alguma tão voláteis como os respectivos programas, tendem ao longo do seu tempo de vida, a ser modificadas, principalmente com a inclusão de novos mecanismos. Obviamente que a extensibilidade nas linguagens depende essencialmente da simplicidade da linguagem base, mas a estrutura e a semântica dos mecanismos a serem adicionados à mesma, é também determinante. De qualquer forma, a ortogonalidade dos mecanismos pré-existentes e dos que se pretende adicionar será o caminho para se maximizar este critério.

Meyer [Meyer 92, Anexo B] apresenta ainda mais dois critérios a se ter em conta.

Unicidade: As linguagens de programação devem fornecer uma boa forma de expressar cada operação de interesse; devem evitar fornecer duas.

Consistência: As linguagens de programação devem assentar num conjunto pequeno de ideias fundamentais e completas, devendo depois as realizar consistentemente até às últimas consequências.

Capítulo 3

Programação e Linguagens (Sequenciais) Orientadas por Objectos

Com este capítulo pretende-se atingir três objectivos:

- apresentar a programação sequencial orientada por objectos;
- enumerar as propriedades e os mecanismos de linguagem que a suportam;
- analisar as interdependências e eventuais interferências entre esses mecanismos.

Não sendo a programação por objectos dissociável do paradigma de programação que a precedeu[1] – a programação procedimental estruturada – far-se-á previamente uma apresentação deste paradigma. Veremos que existem algumas propriedades da programação procedimental estruturada que se mantêm na programação por objectos, e que devem ser tidas em conta quanto a possíveis sinergias e interferências entre mecanismos.

Diferentes linguagens tendem a utilizar diferentes terminologias para os mesmos conceitos e mecanismos, pelo que neste capítulo se irá continuar a estabelecer os termos e definições utilizados nesta tese (os mais importantes e os que mais se prestam a confusões foram também incluídos no glossário).

3.1 Sistemas de tipos

Como foi brevemente referido no capítulo anterior, uma das opções de construção de linguagens mais importante para maximizar a sua segurança assenta no sistema de tipos.

Em linguagens, os "tipos" descrevem a forma e as propriedades dos elementos de um programa que podem estar associados a valores (no caso das linguagens orientadas a objectos puras (página 20) esses valores reduzem-se a objectos). O sistema de tipos, por sua vez, para além de associar – explícita ou implicitamente – os tipos a todos os

[1] Ambas são imperativas.

elementos de software relevantes, verifica (na medida das suas possibilidades) se estes são utilizados correctamente.

Neste trabalho iremos designar por **entidades com tipo**, os elementos sintácticos de uma linguagem que estão associados a um "tipo" (ou seja, em linguagens orientadas por objectos, essas entidades podem conter objectos ou referências para objectos). Dependendo das linguagens podem existir diferentes entidades com tipo, como sejam: variáveis locais, atributos de classes, funções, argumentos formais de rotinas, etc..

Os sistemas de tipos podem ser estáticos[2], dinâmicos ou mistos – consoante a verificação dos tipos é feita, respectivamente, em tempo de compilação, em tempo de execução ou em ambas.

Os sistemas de tipos servem diferentes propósitos [Bruce 02, página 7] [Pierce 02, páginas 4–8]:

- **Segurança**: um sistema de tipos previne a ocorrência, em tempo de compilação ou em tempo de execução, de um conjunto importante de usos incorrectos de entidades com tipo, tais como a aplicação de operações inexistentes. Desta forma melhora-se a segurança da linguagem e a correcção dos programas.

- **Abstracção**: o uso de tipos para anotar as entidades que manipulam valores, possibilita uma separação entre a utilização e a implementação dos valores, o que melhora substancialmente a modularidade do software.

- **Documentação**: os tipos, quando expressos explicitamente, servem também para tornar claras as intenções do programador, podendo assim melhorar bastante a compreensibilidade da linguagem e do software.

- **Optimização**: a verificação de tipos pode fornecer, para o sistema de compilação ou o interpretador da linguagem, informação útil para a geração de código mais eficiente.

Os sistema de tipos estáticos, se comparados com os dinâmicos, melhoram todos estes aspectos. A segurança é substancialmente melhorada já que os erros de tipos são detectados mais cedo, em tempo de compilação. A abstracção e a documentação associadas aos tipos, estando definidas estaticamente, tornam bastante mais claro o propósito de cada tipo sem ser necessário analisar o seu comportamento dinâmico. Por fim, a informação disponibilizada pelos sistemas de tipos estáticos ao compilador, abre a possibilidade de melhorar substancialmente a eficiência dos programas, não só evitando testes de tipos em tempo de execução, como também utilizando técnicas de optimização agressivas (como por exemplo, substituindo uma invocação de uma rotina pelo respectivo código).

No entanto, os sistemas estáticos também podem ter algumas desvantagens. As mais importantes destas são:

- **Tratabilidade**: para que seja possível o sistema de tipos fazer o seu trabalho em tempo de compilação, torna-se necessário que ele seja realizável, ou seja que a sua complexidade não aumente exponencialmente com a dimensão dos programas. Assim, não parece ser em geral possível ter sistemas de tipos estáticos que

[2] *static*

garantam a correcção total do software. Geralmente os sistemas de tipos ficam-se pela verificação de que os valores são conformes com o tipo dos elementos de software que os manipulam[3]. Por esta razão, estes sistemas tendem a ser conservadores, podendo rejeitar programas que, em tempo de execução, nunca teriam comportamentos inseguros.

- **Flexibilidade**: a imposição de que as entidades de um programa só podem conter valores que respeitem o seu tipo – e caso o sistema de tipos seja limitado e pouco expressivo – pode ser um obstáculo substancial à reutilização e à produtividade do software.

O maior problema dos sistemas de tipos estáticos é a necessidade destes dependerem grandemente da forma (sintáctica) do valores, e não do seu comportamento essencial completo (semântico)[4].

Para reduzir substancialmente os problemas de flexibilidade dos sistemas estáticos, ir-se-á mais à frente analisar duas formas essenciais de tornar estes sistemas mais expressivos: polimorfismo de subtipo (secção 3.8) e o polimorfismo paramétrico (secção 3.10).

Neste trabalho, a escolha de linguagens com sistemas de tipos estáticos foi uma opção de base, e mostrou ser uma escolha essencial para os resultados obtidos. No entanto, é importante que não se perca de vista que os sistemas de tipos estáticos não são uma garantia de correcção, mas tão só uma aproximação nesse sentido.

3.2 Programação procedimental estruturada

A programação procedimental parte da ideia base de se expressarem as soluções para problemas como sequências de acções (comandos) a serem executadas. Num programa correcto, à medida que as acções vão sendo executadas, o estado do sistema tende para a solução do problema (essa solução pode estar explicitamente expressa em variáveis do programa, ou implicitamente registada no caminho de execução de comandos que o programa percorre).

Com este método, o problema de programação "reduz-se" – para além de uma especificação adequada (e suficiente) de variáveis para armazenamento explícito de informação do programa – à decomposição de "cima-para-baixo" do algoritmo do procedimento inicial, numa sequência de acções mais simples[5] – podendo elas próprias serem novos procedimentos, passíveis de uma nova decomposição – envolvendo quando necessário instruções de atribuição de valor a variáveis, instruções condicionais[6] e instruções repetitivas[7]. Este processo de decomposição aplica-se hierarquicamente a cada acção resultante da decomposição anterior, até que o algoritmo resultante esteja completamente expresso em função de acções pré-existentes [Wirth 71, Wirth 74].

[3]Esta característica é importante na escolha e comparação entre diferentes aproximações ao polimorfismo subtipo como se verá à frente (página 23)

[4]Veremos (página 24) que a linguagem EIFFEL tem um sistema de tipos que permite, embora de uma forma limitada, que a semântica dos tipos faça parte destes.

[5]Decomposição por "concatenação" segundo Dijkstra [Dijkstra 72, página 19].

[6]Decomposição por "selecção" segundo Dijkstra [Dijkstra 72, página 19].

[7]Estes elementos algorítmicos são suficientes para expressar qualquer algoritmo computável [Böhm 66].

```
if CONDITION then
  COMMANDS
end
```

```
while CONDITION do
  COMMANDS
end
```

```
repeat
  COMMANDS
until CONDITION
```

Figura 3.1: Instruções condicionais e repetitivas estruturadas.

Uma característica importante desta aproximação – aliás partilhada pela programação orientada por objectos – é a sua natureza imperativa. A expressão de um algoritmo é feita por uma sequência de comandos que podem modificar explicitamente o estado do sistema (ou seja a execução de comandos pode ter efeitos colaterais no programa como resultado da modificação do valor das variáveis).

Outro aspecto essencial desta aproximação é a utilização da chamada abstracção algorítmica. Este tipo de abstracção consiste no encapsulamento de algoritmos dentro de procedimentos (acções) ou de funções (cálculo de valores)[8], separando dessa forma a utilização – geralmente simples e facilmente compreensível – da implementação desse algoritmo. Assim, a reutilização de algoritmos e a compreensibilidade dos programas pode ser substancialmente melhorada.

A compreensão de programas será tanto mais facilitada quanto maior for a proximidade entre a sua estrutura estática e o seu comportamento dinâmico (ou seja: em tempo de execução) [Dijkstra 68c]. Uma aproximação nesse sentido será fazer com que as instruções das linguagens tenham apenas um ponto de entrada e um ponto de saída [Dijkstra 72, páginas 16–23] [Wirth 74]. Dessa forma elas podem facilmente ser isoladas e interpretadas como sendo uma única acção numa computação sequencial. Esta propriedade da programação procedimental estruturada é muito importante já que facilita a análise e compreensão de algoritmos de "cima-para-baixo". Assim, as propriedades (que podem ser expressas por axiomas sobre o estado do programa) de cada instrução são definidas de "fora-para-dentro", e não o inverso. É o caso das instruções condicionais e repetitivas estruturadas, cujo comportamento é imposto pela estrutura externamente visível das próprias instruções (figura 3.1).

Os comandos `COMMANDS` – quaisquer que eles sejam – só serão executados caso sejam seleccionados pelas instruções condicionais ou repetitivas onde estão inseridos. Iremos designar as instruções de linguagens que cumpram esta propriedade por instruções estruturadas puras.

Esta propriedade facilita a associação a qualquer acção sequencial A de duas asserções[9] – P e R – atestando a sua correcção [Hoare 69]:

$$\{P\}\ A\ \{R\}$$

Esta fórmula, conhecida por terno de Hoare, pode ser expressa da seguinte forma: se a pré-condição P se verificar no início da execução da acção A, então a pós-condição R será verdadeira no seu fim[10]

[8] Há linguagens, como por exemplo o C que não distinguem de uma forma sintácticamente explícita procedimentos de funções, embora – mesmo nesse caso – se possa considerar que funções do tipo **void** correspondem a procedimentos.

[9] Predicados.

[10] Hoare apresenta esta fórmula com as chavetas a envolver a acção em vez de envolver as asserções: $P\ \{A\}\ R$.

```
     i = 1;             // (1)
l1:  printf("%d\n",i); // (2)
     i++;               // (3)
     if (i <= 10)      // (4)
       goto l1;         // (5)
```

Figura 3.2: Exemplo de um algoritmo com "saltos" em C.

Esta aproximação axiomática à correcção de programas – devida principalmente a Floyd [Floyd 67] e Hoare [Hoare 69] – será uma das contribuições mais importantes da programação procedimental estruturada (tendo sido adaptada e extendida na programação orientada por objectos, com a programação por contrato).

Uma consequência quase imediata desta aproximação à construção de algoritmos é a inadequação da utilização de instruções de "saltos"[11]. Em geral, a utilização de "saltos" torna mais difícil relacionar o comportamento dinâmico de um programa com a sua estrutura textual estática. Essa instrução pode esconder estruturas algorítmicas essenciais como as estruturas repetitivas ou as condicionais muito longe da sua real ocorrência, o que pode tornar o algoritmo de muito difícil compreensão[12] (por essa razão é usual designar a utilização de "saltos" em programas como código tipo "esparguete"). Ou seja, a construção de algoritmos com "saltos", ao contrário das instruções estruturadas puras, pode obrigar à compreensão do algoritmo de "dentro-para-fora".

A figura 3.2 exemplifica a implementação de um algoritmo repetitivo utilizando uma instrução de "saltos". Assim só em `(5)` é que o programador se pode aperceber de que está perante um algoritmo repetitivo iniciado em `(2)`. Muito embora se possam utilizar disciplinadamente as instruções de "saltos" (sendo Knuth o grande defensor dessa utilização regrada [Knuth 74]), tal opção faz com que deixe de haver a garantia em tempo de compilação de que a estrutura algorítmica é simples, perdendo-se a garantia do uso exclusivo de instruções estruturadas puras.

3.2.1 Limitações

A programação procedimental estruturada começa a mostrar as suas limitações à medida que a complexidade do problema a resolver vai aumentando. Com efeito para problemas com alguma complexidade não fará muito sentido atribuir importância a um único procedimento de topo. Facilmente se podem definir vários procedimentos de topo – provavelmente com decomposições de "cima-para-baixo" bastante diferentes – para o mesmo problema a resolver, podendo estes depender, por exemplo, do tipo de interacção entre o utilizador e o programa (interface gráfica, consola de texto, etc.). Fazer depender a decomposição algorítmica dessa escolha conjuntural é claramente um

Estes dois formalismos diferem apenas do detalhe de na notação original de Hoare a pós-condição só ser aplicável caso a acção termine (correcção parcial) enquanto que a notação utilizada pressupõe e impõe a terminação (em tempo finito) da acção [Gries 81, página 109]. Para os objectivos deste trabalho, no entanto, essa diferença não nos parece ser de todo relevante.

[11] *goto*.

[12] Como em todas as regras, há no entanto algumas excepções. Em linguagens sem mecanismos de excepções, o uso de "saltos" pode ser justificado para lidar com situações excepcionais por forma a não "poluir" o código normal e a simplificar programas.

erro e uma sobre-especificação.

Outro problema mais crítico assenta no facto desta aproximação ter uma modularidade fraca. Em geral, os procedimentos e funções não são auto-suficientes, tendo a necessidade de estar associados a estruturas de dados apropriadas. Por exemplo, uma função que indique se uma qualquer data (definida por dia, mês e ano) é válida, está intimamente ligada à estrutura de dados que representa datas (que poderá ser composta por três valores inteiros, por uma estrutura com três campos inteiros, ou uma outra representação qualquer). Uma qualquer modificação da estrutura de dados implica com grande probabilidade a modificação dos procedimentos e funções que dela dependem.

Dos cinco critérios de modularidade apresentados (página 7), três são directamente colocados em causa com esta aproximação:

- *Composição modular*: cada módulo terá de estar ligado aos tipos de dados que utiliza (os quais, por sua vez, podem ter uma coesão grande com outros módulos).
- *Compreensão modular*: a compreensão de cada módulo passa também pela compreensão dos tipos de dados a ele associados, quando não passa também pela compreensão de outras funções (módulos).
- *Protecção modular*, há uma coesão grande com tipos de dados externos ao módulo.

Estas deficiências de modularidade na metodologia da programação procedimental estruturada, são abordadas e resolvidas na programação orientada por objectos.

3.3 Programação por objectos

A denominação "orientado por objectos" tem sido usada e abusada desde que lhe foi atribuído o mesmo estatuto de qualidade que em tempos pertenceu à programação (procedimental) estruturada. Na realidade constata-se que diferentes escolas de programação – geralmente intimamente ligadas a diferentes linguagens – têm uma percepção diferente do que constitui este tipo de programação. Aliás, o autor desta tese também não é completamente imune a este problema, sendo defensor de uma abordagem em particular à programação orientada por objectos assente em muitos dos princípios que estão na base do método e linguagem Eiffel. Não obstante esta possível limitação, iremos tentar apresentar não só as propriedades que quase consensualmente são atribuídas às linguagens orientadas por objectos, como também outras propriedades e mecanismos considerados importantes.

Embora se deva separar os conceitos de programação por objectos (metodologia) das linguagens de programação em função das quais os programas são expressos, nesta secção ir-se-á misturar um pouco esses dois mundos. Esta opção (sem dúvida discutível), é justificada pelo autor pelo facto deste trabalho incidir essencialmente em linguagens de programação, e muito em particular na perspectiva de que estas podem contribuir decisivamente para a correcção do software. Ora o reforço da correcção de programas depende fortemente da metodologia de programação seguida, pelo que se pode considerar que ambos os mundos se unem para o mesmo fim. Esta aproximação é uma vez mais influenciada pela linguagem Eiffel que é apresentada pelo seu autor como não apenas uma linguagem mas também um método de programação.

Primeiramente vamo-nos debruçar sobre os seis mecanismos e propriedades essenciais que julgamos mínimas para definir quer as linguagens quer a própria programação por objectos. Seguidamente abordaremos outros mecanismos frequentemente utilizados em linguagens por objectos, muitos deles desejáveis pelo impacto positivo que poderão ter na qualidade quer dos programas quer das linguagens; outros indesejáveis pela razão oposta.

3.4 Objecto: estrutura de dados + métodos

Uma primeira aproximação à programação orientada a objectos resulta de duas constatações (complementares) retiradas da análise feita à programação procedimental e das quais se retira o mesmo resultado. A primeira constatação é que muitos métodos (funções e procedimentos) tendem a estar íntima e fortemente ligados a determinadas estruturas de dados. Uma mudança na estrutura de dados implica muitas vezes a modificação, parcial ou mesmo total, dos métodos que dela dependem directamente. Por outro lado, analisando o problema do lado das estruturas de dados, estas por si só, são entidades passivas cujo comportamento (semântica) lhes é, em grande medida, imposto exteriormente precisamente pelos métodos que directamente as manipulam. Por exemplo uma estrutura de dados com três campos inteiros, tanto pode servir para representar uma data (dia, mês e ano) como um relógio (horas, minutos e segundos) ou qualquer outra "coisa" envolvendo três valores inteiros. No entanto o comportamento em cada uma dessas possibilidades será bastante diferente (e incompatível entre si). Não fará muito sentido atribuir o valor `15` a um mês, nem `2006` aos segundos de um relógio.

Assim sendo, parece haver vantagem quer na perspectiva dos métodos, quer na das estruturas de dados, em juntar ambos numa única entidade. A essa entidade, na programação por objectos, é dado o nome de **objecto**.

Os elementos das estruturas de dados que definem e permitem armazenar o estado do objecto são habitualmente designados por **atributos** (o seu comportamento dentro de cada objecto, é similar ao das variáveis das linguagens procedimentais). Esses atributos podem ser variáveis ou constantes. Iremos também indistintamente designar por **serviços** (*features* na terminologia utilizada na linguagem Eiffel) o conjunto de métodos (que trataremos também por rotinas) e atributos aplicáveis a objectos. Assim, um objecto é constituído por um conjunto de serviços, podendo estes ser atributos ou métodos. Quando se justificar, poder-se-á ainda dividir os métodos em funções e procedimentos. As **funções** são abstracções algorítmicas de observação ou consulta sobre o estado do objecto. Os **procedimentos** são abstracções algorítmicas de comandos aplicáveis ao objecto por forma a modificar o seu estado. As funções que não tiverem efeitos colaterais no estado observável do objecto, nem no estado observável de nenhum outro objecto do programa, serão designadas por "puras". Outra classificação muito útil dos serviços de objectos é a separação entre comandos (*commands*) e consultas (*queries*). Os comandos de um objecto serão os respectivos procedimentos, enquanto que as consultas serão os seus atributos e funções (que devem preferencialmente ser puras).

Ao contrário das rotinas e das estruturas de dados – que necessitam e dependem

uma da outra – os objectos são auto-suficientes para a construção de programas. Assim é possível definir linguagens de programação orientadas por objectos em que todo o programa é exclusivamente construído à custa de objectos. Estas linguagens designam-se por linguagens orientadas por objectos puras.

3.5 Objectos e classes

Existem basicamente duas aproximações linguísticas à construção e instanciação de objectos. Na primeira, o comportamento dos objectos é definido separadamente em entidades sintácticas designadas por classes[13], sendo cada objecto criado como uma instância de uma classe. Nesta aproximação as classes são também a base para definir os tipos dos objectos. A segunda aproximação assenta em protótipos [Borning 86, Lieberman 86, Ungar 91]. Um objecto é criado directamente a partir de uma descrição do conjunto de métodos e atributos desejado, ou clonando e adaptando um outro objecto (protótipo).

A larga maioria das linguagens orientadas a objectos seguem a primeira aproximação: Simula [Dahl 68], Smalltalk [Goldberg 89], Eiffel [Meyer 92], C++ [Stroustrup 97], Java [Gosling 05], CLOS [Bobrow 88], BETA[14] [Madsen 93]. Há no entanto um (pequeno) grupo de linguagens assente em protótipos: Self [Ungar 87], Cecil [Chambers 04].

Este trabalho incide apenas sobre linguagens orientadas a objectos baseadas em classes.

3.6 Encapsulamento de informação

O encapsulamento de informação (devido a David Parnas [Parnas 72b, Parnas 72a]) em objectos é a possibilidade destes esconderem um subconjunto dos seus serviços dos seus utilizadores externos.

Embora não exista esta possibilidade na linguagem considerada como a fonte da programação por objectos – a linguagem SIMULA –, e de na linguagem Smalltalk (onde pela primeira vez apareceu a designação "orientado por objectos") o encapsulamento ser pré-definido pela linguagem (os atributos são sempre privados e os métodos públicos); poucas dúvidas existem actualmente quando à importância essencial do encapsulamento de informação para a programação orientada por objectos.

O encapsulamento de informação vai de encontro a três dos critérios de modularidade apresentados anteriormente (página 7):

- *Compreensão modular*: um objecto pode ser compreendido (e utilizado) apenas tendo em conta o subconjunto de métodos (mais à frente abordaremos o problema dos atributos públicos) considerado essencial.

[13] Apesar das classes serem entidades sintácticas, há linguagens, como o Smalltalk que permitem a sua modificação em tempo de execução.

[14] Esta linguagem permite também a criação de objectos sem classes.

- *Continuidade modular*: os métodos e atributos que não são visíveis do exterior podem ser retirados ou modificados livremente sem que se corra o risco de afectar directamente os clientes do objecto.

- *Protecção modular*: existe a possibilidade de os objectos poderem ser os únicos responsáveis no controlo da correcção do seu estado interno, prevenindo a ocorrência de usos incorrectos (como por exemplo, definir o dia `32` num objecto `DATA`).

Da discussão feita, em particular no que diz respeito à protecção modular, podemos concluir que um objecto não deve ter atributos que possam ser directamente modificáveis por clientes (públicos na terminologia das linguagens C++ e JAVA). Nessa situação, não só o objecto deixa de poder controlar a sua própria correcção, como também liga directamente a sua interface a uma escolha em particular de representação do seu estado (sobre-especificação).

O encapsulamento de informação tem um efeito directo nos seguintes factores de qualidade: correcção, extensibilidade, reutilização, verificabilidade e compreensibilidade.

3.7 Herança

Outro mecanismo considerado essencial da programação por objectos (assente em classes) é a chamada "herança". Este mecanismo permite construir novas classes a partir de outras pré-existentes, reutilizando e eventualmente redefinindo métodos e atributos.

Uma classe ao herdar de outra (classe ascendente ou super-classe), automaticamente passa a ter todos os seus métodos e atributos, tendo a possibilidade de redefinir alguns destes caso tal seja necessário ou tão só conveniente. Desta forma a herança promove um estilo de programação por diferença – possibilitando a construção de novas classes à custa de outras pré-existentes – minimizando assim a redundância de código e aumentando as possibilidades de reutilização.

Quando uma classe `A` herda de outra classe `B`, diz-se que `A` é uma **subclasse** ou classe descendente de `B`. Meyer [Meyer 97, página 464] generaliza a definição fazendo com que um descendente de uma classe seja a própria classe ou um qualquer dos seus herdeiros directos ou indirectos.

3.7.1 Encapsulamento de informação

Um aspecto importante – e para o qual se encontram aproximações diferentes em diferentes linguagens – tem a ver com a interferência entre herança e encapsulamento de informação.

Por um lado, levanta-se a questão de dever haver, ou não, encapsulamento de informação relativamente à subclasse. Algumas linguagens (por exemplo: C++ e JAVA com os serviços privados) permitem esse encapsulamento. Outras (EIFFEL), não dão essa possibilidade.

Por outro lado, coloca-se também o problema de até que ponto as subclasses podem redefinir o encapsulamento de informação existente na classe ascendente (ou classes

ascendentes, no caso de haver herança múltipla). Também aqui a aproximação difere consoante as linguagens consideradas. Em C++ e JAVA uma subclasse apenas pode manter ou restringir mais o encapsulamento de informação da classe ascendente. Já em EIFFEL, há uma completa ortogonalidade entre os dois mecanismos. Meyer [Meyer 97, página 57] sustenta esta opção, recorrendo ao chamado princípio de modularidade "Aberto-Fechado".

> Um módulo deve estar simultaneamente aberto e fechado.

Este principio defende que um módulo deve estar aberto a ser modificado e adaptado a novas situações e necessidades, e fechado para poder ser utilizado com segurança por clientes. O "truque" para se conseguir conciliar este aparente paradoxo assenta precisamente no mecanismo de herança (um módulo deverá estar aberto a ser apropriadamente modificado em subclasses) e na ortogonalidade deste (também) relativamente ao encapsulamento de informação[15].

3.8 Polimorfismo de subtipo e encaminhamento dinâmico (simples)

Diz-se que um do tipo `T` é um **subtipo** (conforme, na terminologia da linguagem EIFFEL) de um tipo `U` (`T <: U`) se um objecto do tipo `T` puder ser utilizado em todos os contextos onde se espera objectos do tipo `U`.

Esta possibilidade de a uma entidade do tipo `U` poder estar associada a um objecto de um subtipo é designada por **polimorfismo de inclusão ou subtipo** [Cardelli 85].

Para que seja possível associar a uma entidade `target` do tipo `T` um objecto `obj` de um qualquer subtipo `U`, é necessário que a invocação de um qualquer serviço através de `target` seleccione o serviço apropriado do `obj` em `U`. Se uma mesma entidade `target` puder estar associada em tempo de execução do programa, a objectos de tipos diferentes, então essa selecção terá de ser feita dinamicamente, consoante o tipo do objecto ao qual `target` está associado. Nas linguagens orientadas por objectos, essa escolha é feita pelo próprio objecto através de um mecanismo denominado por **encaminhamento dinâmico simples** (na literatura aparecem várias designações para o mesmo mecanismo, como seja: *dynamic binding*, ou *simple dispatch*).

Esta característica essencial de, nas linguagens orientadas por objectos, o serviço a executar ser seleccionado pelo próprio objecto, justifica o uso para invocação de serviços de objectos, da designação alternativa (mas equivalente) de envio de mensagens utilizada sobretudo nas linguagens da família do SMALLTALK.

Os mecanismos de polimorfismo subtipo e encaminhamento dinâmico, permitem aumentar tremendamente a flexibilidade do sistema de tipos estático[16], sem o comprometer[17].

[15] No entanto, esta liberdade pode levantar alguns problemas, tais como a garantia de substitutabilidade, como seguidamente se verá.

[16] Mais à frente na secção 3.10 será apresentado outro mecanismo de polimorfismo – designado de paramétrico – que aumenta ainda mais as garantias de correcção em tempo de compilação do sistema de tipos.

[17] Os problemas relacionados com a herança serão analisados à frente (página 24).

Embora eventualmente possa fazer algum sentido falar de subtipos em sistemas de tipos dinâmicos – uma vez que nestes se pode geralmente tentar fazer passar um qualquer objecto por outro, sendo a substitutabilidade verificada dinamicamente mensagem a mensagem, e não para o tipo completo do objecto – é nos sistemas de tipos estáticos que essa relação é mais importante, e onde também é colocado o desafio mais difícil de como expressar de uma forma segura a relação subtipo.

3.8.1 Escolha dinâmica de rotinas *versus* escolha dinâmica de objectos

É interessante comparar-se esta aproximação orientada por objectos – em que é o próprio objecto que determina dinamicamente o serviço a ser executado – com a aproximação procedimental (e também funcional), em que é a rotina a determinar dinamicamente (com uma instrução de selecção múltipla) qual o tipo de objecto a qual está a ser aplicada. Apesar de as duas aproximações serem duais, a escolha entre ambas não é em geral de todo indiferente. As estruturas de dados tendem a ser bastante mais estáveis do que as rotinas, pelo que acrescentar novos serviços a classes tende a ter menos efeito na modularidade do programa do que acrescentar novos tipos de dados a funções (sendo aproximações duais, estamos a comparar as extensões também duais em ambas as aproximações). Por outro lado – graças à herança – as classes não necessitam de implementar (ou mesmo muitas vezes sequer conhecer) todos os seus serviços (a programação por diferença mostra aqui o seu poder). Já a aproximação procedimental, a não existir um mecanismo de herança aplicável às rotinas similar ao das linguagens orientadas por objectos, obriga a que todas essas rotinas conheçam os tipos de dados a que são aplicadas.

Vemos assim que as duas aproximações têm um impacto muito diferente no critério de modularidade da continuidade (página 7).

As chamadas implementações convencionais de tipos de dados abstractos[18] – existente, por exemplo, nas `packages` da linguagem Ada [Ada95 95], nos módulos da Modula-2 [Wirth 85] e nos *clusters* da linguagem CLU [Liskov 77] – seguem também a aproximação procedimental apresentada, sobrecarregando as rotinas com a escolha interna sobre qual é o tipo de dados (a representação do tipo de dados abstracto) ao qual está a ser aplicada.

3.8.2 Relações de subtipo nominais e estruturais

No que diz respeito às relações de subtipo, a herança não é a única possibilidade de as expressar. De facto, podem-se identificar duas formas distintas de expressar essa relação em linguagens de programação: ou de uma forma explícita (nominal), ou de uma forma implícita (estrutural). Na primeira – que é de longe a mais frequente em linguagens orientadas a objectos (Eiffel, Java, C++) – a relação de subtipo é expressa explicitamente através de um mecanismo de linguagem adequado, geralmente o mecanismos de herança[19] (subclasse). Na segunda forma (existente por exemplo em Emerald [NC 87]) – mais frequente em linguagens de programação mais orientadas para a programação funcional – a relação de subtipo é implícita e garantida sempre que

[18] Os tipos de dados abstractos são apresentados mais à frente (página 26).

[19] Em Java, para além das classes, essa relação pode também ser expressa por interfaces.

o subtipo partilha (pelo menos) a mesma estrutura (nomes e assinaturas) do super-tipo (chama-se a essa propriedade: equivalência estrutural).

Ambas as aproximações têm vantagens e desvantagens. A aproximação estrutural, tem a vantagem de poder ser facilmente extendida com super-tipos, sem que tal afecte minimamente os respectivos subtipos. Desta forma é facilitada a redefinição do grafo de subtipos do programa, sem ser necessário mexer nos tipos existentes. Outra vantagem desta aproximação, é possibilidade (bem documentada na literatura [Cardelli 85, Pierce 02, Bruce 02]) de implementar sistemas de tipos estáticos seguros e tratáveis, onde é garantida a segurança (estrutural) de subtipos. No entanto, esta aproximação, tem duas grandes desvantagens. A primeira é o facto de a relação de subtipo entre tipos ser (por definição) implícita e casual, não resultando de uma opção explicitamente tomada pelo programador. Assim, facilmente um subtipo pode deixar de o ser, ou *vice-versa*, apenas por uma mudança na forma dos seus serviços. A segunda, e sem dúvida a mais importante, tem a ver com o significado e utilidade dos tipos na concepção de programas. Utilizando a definição atrás apresentada (página 13), os tipos descrevem a forma e as propriedades das entidades que podem estar associadas a valores num programa. Ora uma aproximação por equivalência estrutural aos subtipos restringe drasticamente a possibilidade de associar e impor propriedades na relação de subtipo, para além daquelas óbvias que têm apenas a ver com a estrutura formal dos tipos (nomes e assinaturas dos serviços).

Com equivalência estrutural é perfeitamente possível que um tipo correspondente a um `STACK` seja substituível por outro correspondente a uma `QUEUE`, bastando para tal que ambos partilhem a mesma estrutura (o que é frequente), embora – como é óbvio – esses tipos não sejam de todo substituíveis, já que têm um comportamento distinto e incompatível. A aproximação ao polimorfismo subtipo recorrendo à herança tem, neste aspecto, a vantagem de garantir que só são substituíveis objectos que sejam descendentes explícitos de um determinado tipo. No caso da linguagem Eiffel, essa vantagem é ainda maior já que as propriedades semânticas das classes são obrigatoriamente herdadas em classes descendentes (secção 3.12).

3.8.3 Segurança

Apesar dessas vantagens, a relação directa de subtipos com herança, pode gerar problemas de segurança estática no sistema de tipos. É o que pode acontecer quando se permite a mudança na visibilidade externa de serviços (fazendo com que, por exemplo, um serviço público na classe pai, passe a ser privado na classe descendente); ou quando se permite a redefinição covariante[20] de entidades com tipo que possam ser destinos de atribuições de valor (*left-values* na terminologia da linguagem C) [Bruce 02].

Este problema – embora possa colocar problemas de segurança sérios – sai fora do âmbito deste trabalho. A linguagem Eiffel tem este problema, existindo várias propostas para o resolver, seja obrigando o sistema de tipos a uma análise global dos programas (validação do sistema [Meyer 97, página 633]); proibindo a existência de *catcalls*[21] polimórficos (ou seja proibindo o uso de polimorfismo subtipo sobre serviços

[20] Ou seja, no mesmo sentido da relação de herança.

[21] *Change Availability of Type calls*

covariantes); ou mesmo à separação entre os mecanismos de herança e de subtipo[22] [Cardelli 88, Cook 90, Bruce 93].

Uma outra alternativa que julgamos poder ser válida consiste em acrescentar à linguagem um mecanismo de encaminhamento dinâmico múltiplo orientado por objectos[23].

3.8.4 Subclasse *versus* subtipo

Neste trabalho iremos pressupor não só a relação explícita de subtipos, como também se irá considerar que uma relação de subclasse (herança) implica uma relação de subtipo[24].

Até que ponto é que essa aproximação será aceitável? São bem conhecidas na literatura várias referências muito críticas relativamente a essa ligação [Cook 90][Bruce 02, páginas 24-26], essencialmente como resultado dos problemas de segurança já referidos.

A herança é – por definição – um mecanismo de reutilização. Dito de outra forma, uma classe ao herdar de outra (ou de outras) deveria ser absolutamente equivalente a uma outra classe que directamente implementasse os serviços dessa classe pai. Por outro lado – uma vez que a classe está a reutilizar os serviços da classe pai – no caso geral terá todas as possibilidades de cumprir o mesmo contrato (ou seja de respeitar o mesmo TDA[25]) do que essa classe ascendente. Excepcionalmente – pela razões já apresentadas – tal poderá não acontecer, mas a regra será o cumprimento integral. Ou seja: em regra uma relação subclasse tem todas as condições para ser considerada uma relação subtipo.

Porquê então impor uma separação entre os dois mecanismos, quando – para a maior parte dos casos – tal irá obrigar ao uso duplicado de ambos, nas relações entre classe pai-filho?

Será talvez defensável ter um mecanismo separado para esse casos excepcionais (como a recente proposta de herança não conforme para a linguagem Eiffel), mas seria um erro enorme fazer com que todas as relações de subclasse não fossem também relações de subtipo (em Java, embora possuindo apenas herança simples, as relações de subclasse implicam também uma relação de subtipo).

3.9 Objectos e tipos de dados abstractos

As características consideradas como essenciais a existir em linguagens orientadas por objectos serão as cinco anteriores: objectos, classes, encapsulamento, herança e subtipos. No entanto, falta ainda um suporte teórico que permita descrever de uma forma apropriada os objectos, e que não só inclua todos esses mecanismos como também lhes dê coerência, consistência e sentido. Esse é o papel dos *tipos de dados abstractos*.

Liskov e Zilles [Liskov 74] definiram originalmente um Tipo de Dados Abstracto (TDA) como sendo:

[22] Opção que nos parece ir no caminho errado.

[23] O que, na nossa opinião, exclui a aproximação de multi-métodos da linguagem CLOS.

[24] Esta é a opção feita na linguagem Eiffel, embora recentemente esteja a ser considerada a inclusão de um mecanismo de subclasse que não implica subtipo [ECMA-367 05, página 16].

[25] Tipo de Dados Abstracto (página 26)

Uma classe de objectos abstractos que são completamente caracterizados pelas operações existentes sobre esses objectos.

No entanto, esta definição não é completamente satisfatória. Se um TDA for encarado como sendo definido apenas pelos nomes e assinaturas das operações que lhe são aplicáveis, então – tal como acontece com a aproximação estrutural à relação de subtipo atrás referida (página 23) – facilmente se pode ter o mesmo TDA para abstracções diferentes e incompatíveis (insubstituíveis) [Guttag 77]. Por exemplo, um TDA para uma "pilha" (*STACK*) pode ser definido formalmente da seguinte forma (adaptado de [Meyer 97, página 139]).

TYPES
$STACK[T]$

FUNCTIONS
$new : \rightarrow STACK[T]$
$put : T \times STACK[T] \rightarrow STACK[T]$
$remove : STACK[T] \nrightarrow STACK[T]$
$top : STACK[T] \nrightarrow T$
$empty : STACK[T] \rightarrow BOOLEAN$

Esta mesma estrutura pode-se aplicar sem modificações (para além, é claro, do nome do tipo) para "filas" (*QUEUE*), embora, como é evidente, em caso algum objectos que implementem esses TDAs sejam substituíveis entre si[26].

Uma definição mais apropriada e completa de TDA – onde a semântica (definida axiomaticamente) do TDA é explicitamente incluída – é apresentada por Guttag [Guttag 77] e Meyer [Meyer 88b, Meyer 97].

Tipo de Dados Abstracto (TDA)

Uma classe de objectos abstractos que são completamente caracterizados pelas operações existentes sobre esses objectos e pela respectiva semântica.

Os TDAs fornecem um suporte formal sólido para descrever os objectos e as respectivas classes.

Classe

Uma classe é uma implementação possivelmente parcial de um tipo de dados abstracto [Meyer 97, página 142].

[26] Como vimos atrás (página 23) essa é uma das críticas que pode ser feita aos sistemas de tipos que definem a substitutabilidade apenas por equivalência estrutural.

Os TDAs dão também suporte para o encapsulamento de informação, permitindo uma escolha adequada dos serviços de cada classe que devem, ou não, ser públicos [Meyer 97, página 144].

A semântica dos TDA deve ser expressa axiomaticamente associando à classe três tipos de asserções: invariantes, pré-condições e pós-condições. Os invariantes são axiomas que têm sempre de ser verificados em qualquer interacção com as instâncias da classe (ou seja, quando um qualquer dos seus serviços é externamente utilizado). As pré-condições e as pós-condições são definidos para cada serviço da classe, e são aplicados, respectivamente, quando esse serviço é invocado e quando termina a sua execução.

Assim temos que, a qualquer serviço `S` pertencente a uma classe com o invariante `INV`, aplica-se a seguinte condição de correcção [Meyer 97, páginas 368–370]:

$$\{INV \ and \ PRE_S\} \ ROUTINE - BODY_S \ \{INV \ and \ POST_S\}$$

Ou seja, a execução de um qualquer serviço é correcta (relativamente às asserções expressas) se, imediatamente antes do início da sua execução, o invariante da classe a que pertence e a pré-condição desse serviço forem verdadeiros; e se o mesmo acontecer ao invariante e à pós-condição logo após essa execução. Vemos assim que o suporte axiomático para a correcção de serviços assenta na aplicação do terno de Hoare (página 16) aos serviços da classe.

Muito embora a semântica dos TDA se deva sempre aplicar às classes que os implementam, seja qual for a linguagem por objectos utilizada, é extremamente desejável que a própria linguagem suporte a expressão dessa semântica, já que essa aproximação tem um impacto fortíssimo na correcção de programas (para além de afectar positivamente a sua robustez, legibilidade e verificabilidade). Infelizmente poucas linguagens – entre as quais se destaca a linguagem Eiffel – oferecem esse suporte. Na secção 3.12 iremos apresentar a metodologia da programação por contrato que assenta precisamente nesse suporte.

É importante referir-se que embora, com a excepção da linguagem Eiffel, nenhumas das linguagens orientadas por objectos mais conhecidas tenham, de base, mecanismos para expressar estas asserções em classes, tal não significa que estas não deixem de dever ser vistas como implementações (eventualmente parciais) de TDAs. Embora esta perspectiva ainda não seja assumida explicitamente por todos na programação por objectos, na opinião do autor tal será mais ou menos inevitável, dada as vantagens esmagadoras que dela resultam.

Este trabalho irá pressupor (explicitamente) esta visão da programação por objectos, sendo mesmo um dos aspectos onde houve um maior cuidado na integração segura de mecanismos de concorrência em linguagens orientadas por objectos.

Assim será considerada a definição de Meyer para a programação orientada por objectos [Meyer 97, página 147]:

Programação Orientada por Objectos

A programação orientada por objectos é a construção de sistemas de software como colecções estruturadas de implementações, possivelmente parciais, de tipos de dados abstractos.

Uma última nota. Em vez de se utilizar uma definição axiomática, a semântica dos TDAs pode ser expressa de uma forma operacional[27]. No entanto, essa aproximação traz vários problemas [Guttag 77]. Não só gera com facilidade sobre-especificações, como também dificulta a compreensão dos TDAs, reduzindo a sua utilidade. Outro aspecto muito importante a ter em conta é o interesse em se fazer com que a semântica faça parte dos TDA, e – na medida do possível – das suas implementações (aspecto tratado na secção 3.12).

Os mecanismos tidos como essenciais (e mínimos) a existir em linguagens por objectos são os apresentados nestas últimas seis secções. Iremos agora apresentar outros mecanismos que são opções frequentes em muitas linguagens orientadas por objectos. A maioria destes integram-se bem na programação por objectos e contribuem de uma forma importante para a melhoria da sua qualidade.

3.10 Parametrização de tipos: polimorfismo paramétrico

Um mecanismo muito útil é a possibilidade de se especificar classes em função de tipos genéricos (sem a sobre-especificação de ter de escolher apenas um tipo na implementação dessas classes). Por exemplo, o TDA de uma pilha não depende de forma alguma do tipo de elementos que a podem constituir. Assim, faz todo o sentido construir a classe `STACK` parametrizada relativamente ao tipo dos elementos, por forma a se poder criar diferentes tipos de pilhas, como seja uma pilha de números inteiros ou de datas, sem ser necessário criar uma "nova" classe `STACK` para cada um desses tipos de elementos. Mais, é também desejável que se possa conhecer, para cada pilha, qual o tipo em particular partilhado[28] por todos os seus elementos, por forma a estes poderem ser utilizados tirando partido dos respectivos TDAs. Esse mecanismo é designado por polimorfismo paramétrico [Cardelli 85] (a primeira definição e classificação, ainda que incompleta, dos vários tipos de polimorfismo, incluindo o polimorfismo paramétrico, deve-se a Strachey em 1967 [Strachey 00]).

Este mecanismo é relevante em linguagens com sistemas de tipos estáticos. Nas linguagens com sistemas de tipos dinâmicos existe bastante mais flexibilidade na mistura e substitutabilidade de objectos, pelo que a parametrização de classes se faz com facilidade sem a "oposição" do sistema de tipos (o preço a pagar por essa flexibilidade é uma muito menor segurança da linguagem).

Os factores de qualidade de programas afectados positivamente por este mecanismo são a reutilização, a extensibilidade e a correcção (este último da segurança com que este tipo de polimorfismo pode ser implementado em linguagens com sistemas de tipos estáticos).

[27] Como se verá em capítulos posteriores, coloca-se o mesmo dilema na escolha da semântica dos mecanismos concorrentes em linguagens de programação, especialmente no que diz respeito ao sincronismo de objectos concorrentes. Sem surpresa constatar-se-á que a aproximação axiomática é bastante mais simples e segura.

[28] O polimorfismo de subtipo é aplicável pelo que os objectos pode ser de tipos diferentes desde que descendentes do tipo do elemento da pilha especificado.

3.10.1 Relação com o polimorfismo subtipo

Em linguagens orientadas por objectos puras, geralmente todos os objectos são subtipos de um único tipo (em SMALLTALK será o objecto `OBJECT` e em EIFFEL a classe `ANY`). Nesses casos, pode-se simular o polimorfismo paramétrico recorrendo ao polimorfismo de subtipo, bastando para tal que se utilize esse super-tipo comum (ou outro qualquer que seja conveniente) como parâmetro da classe. Dessa forma, essa classe pode ser reutilizada para objectos de outros quaisquer tipos descendentes. Apesar disso, essa opção não é desejável, uma vez que se perde a informação estática do tipo desses parâmetros, o que pode pôr em causa a correcção dos programas.

Assim, embora se possa relacionar os dois tipos de polimorfismo, em sistemas de tipos estáticos seguros, ambos são importantes e geralmente servem propósitos diferentes [Meyer 86].

3.10.2 Polimorfismo paramétrico restringido

Alguns mecanismos de polimorfismo paramétrico permitem, quando desejado, que se imponham restrições aos parâmetros de tipos. Esse tipo de polimorfismo é designado por polimorfismo paramétrico restringido (*bounded*[29]) [Cardelli 85]. Por exemplo, caso se queira construir uma classe para implementar listas ordenadas (a condição de elementos da lista estarem sempre ordenados poderia ser um dos invariantes dessa classe), parametrizada relativamente ao tipo dos seus elementos, torna-se necessário garantir que esta lista só pode ser instanciada com elementos que estabeleçam uma relação de ordem entre eles. Caso exista um terceiro tipo – `COMPARABLE` – com o TDA de relação de ordem (operações `greater-than` e `lower-than`), então pode-se construir a classe lista restringido o tipo dos seus elementos a serem descendentes desse tipo `COMPARABLE`, garantindo assim estaticamente que a classe só será parametrizada com elementos que definam uma relação de ordem entre eles.

No polimorfismo paramétrico restringido pode-se generalizar a condição de restrição imposta aos tipos dos parâmetros fazendo com que ela seja expressa por uma função de tipos, em vez de um tipo constante predefinido. Este tipo de polimorfismo designa-se por polimorfismo paramétrico F-restringido (*F-bounded*) [Canning 89].

3.11 Herança múltipla

A herança simples permite a construção de uma classe à custa de outra pré-existente e, caso também implemente a relação subtipo, define as regras de substituição polimórfica de entidades com tipo do programa. A herança múltipla generaliza este mecanismo, permitindo a construção de classes à custa de mais do que uma classe ascendente.

Este mecanismo não é de forma alguma consensual na comunidade da programação por objectos. A sua má fama é em parte justificada pela aproximação que lhe é feita, por uma das linguagens orientada por objectos mais populares: o C++ (de tal forma

[29] Em EIFFEL utiliza-se o termo *constrained*.

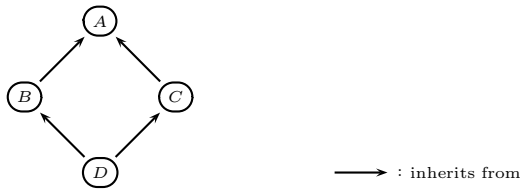

Figura 3.3: Herança repetida.

que justificou a sua não inclusão em JAVA[30]). O facto de as primeiras linguagens por objectos – SIMULA67 e o SMALLTALK – não terem herança múltipla também contribuiu para que esta fosse desde logo encarada com bastante desconfiança.

Uma argumentação recorrente (neste como em muitos outros mecanismos) assenta na possibilidade de se poder simular a herança múltipla com herança simples utilizando, por exemplo, a técnica dos "objectos gémeos" [Moessenboeck 93, Templ 93]. Essa aproximação, no entanto, não só omite o problema da herança repetida (que ocorre sempre que as relações estáticas de herança entre classes não podem ser expressas por uma árvore), como também é uma sobre-especificação deste mecanismo (expressando-o em função de uma possível implementação).

Uma diferença interessante – continuando a assumir que a herança estabelece relações de subtipo – entre a herança simples e múltipla, é a possibilidade de uma classe ser subtipo de duas (ou mais) classes que não se relacionam entre si também por uma relação de subtipo (propriedade que é sempre verificada na herança simples).

Outra propriedade interessante é o facto de as relações estáticas de herança entre classes serem representáveis por um grafo (dirigido), e não necessariamente uma estrutura de dados do tipo árvore.

3.11.1 Herança repetida

Um dos problema – designado por herança repetida – levantados por este mecanismo ocorre sempre que uma classe, directa ou indirectamente, herda mais do que uma vez de uma mesma classe.

A figura 3.3 exemplifica esta situação: A classe `D` herda "duas vezes" da classe `A`. Devem os atributos de `A` ser todos duplicados em `D`; partilhados ou uma mistura criteriosa de ambos os casos? Em C++ só existem as duas primeiras possibilidades havendo partilha ou separação integral quando respectivamente em `B` e `C` a classe `A` é herdada, ou não, virtualmente. Esta aproximação é claramente errada já que obriga a que esta decisão importantíssima para `D` seja tomada nas classes `B` e `C` (e não na própria classe `D`).

Por outro lado existe também o problema da partilha ou não dos restantes serviços de `A` herdados repetidamente em `D` através de `B` e `C`. Novamente em C++ a aproximação tomada é bastante deselegante e problemática. O uso de um serviço da classe `A` que

[30] Onde, no entanto, foi acrescentado um mecanismo – interfaces – para permitir relações de subtipos similares à herança múltipla.

queira utilizar os atributos de `A` herdados por `B` terá de explicitamente indicar essa classe base `B` na invocação desse serviço em `D`.

3.11.2 Colisão de nomes

Outro problema de segurança levantado pela herança múltipla consiste na situação em que a classe herda de duas (ou mais) classes um serviço com a mesma assinatura ou tão só apenas com o mesmo nome. Nessa situação qual dos serviços, se algum, deverá ser seleccionado para execução? Em C++ a situação é agravada pelo facto de esta linguagem permitir a sobrecarga de serviços (secção 3.14) o que pode gerar ambiguidades, por vezes difíceis de detectar e corrigir.

Em EIFFEL todos estes problemas são resolvidos de uma forma extremamente elegante. Nesta linguagem não é permitida a possibilidade de numa classe existirem dois (ou mais) serviços com o mesmo nome[31]. Sempre que uma classe herda um serviço com o mesmo nome de duas ou mais classes, é obrigada a mudar o nome de pelo menos um desses serviços por forma a que um nome corresponda apenas a um único serviço da classe. Este mecanismo de mudança de nome reside na classe onde o problema se coloca, e permite uma solução elegante para a partilha ou replicação de serviços na herança repetida. Voltando a utilizar o exemplo da figura 3.3, haverá partilha dos serviços de `A` se estes forem herdados em `D` com o mesmo nome e caso não tenham sido redefinidos em `B` e `C`, ou replicados no caso contrário[32].

3.11.3 Classes equivalentes

Uma consequência muito importante desta aproximação feita em EIFFEL, é o facto de ela garantir, para qualquer classe, a existência de uma classe absolutamente equivalente construída sem herança[33]. É possível até, caso as classes ascendentes não sejam necessárias em nenhuma parte do programa para eventuais utilizações do polimorfismo subtipo, substituir uma qualquer classe por essa classe equivalente.

Uma incompreensão relativamente frequente quanto ao mecanismo de herança (como por exemplo em [Ryant 97]) na programação por objectos, consiste em considerar que uma instância de uma classe, implementada herdando de classes ascendentes, de alguma forma contém um objecto de cada uma dessas classes[34]. A herança não é um mecanismo de inclusão de objectos, mas sim de partilha do código de classes (relação de subclasse), e de substitutabilidade de objectos (relação de subtipo).

3.12 Suporte para a programação por contrato

A programação por contrato [Meyer 97, página 331] permite completar a implementação prática dos TDA fornecida pelas classes, possibilitando a expressão da res-

[31] Sobrecarga de serviços.

[32] No caso mais complexo de os serviços terem o mesmo nome mas tenham sido redefinidos nas classes intermédias, a linguagem EIFFEL permite mesmo assim em certos casos a junção desses serviços num único, mas não iremos abordar essa situação aqui.

[33] Esta operação é designada por *flat form* em EIFFEL [Meyer 97, página 541].

[34] A aproximação do C++ à herança múltipla será uma das responsáveis por essa confusão.

	Deveres	Direitos
Cliente	Satisfazer a pré-condição de cada serviço requerido.	Garantia que quer o invariante da classe, quer a pós-condição do serviço requerido, se verificam quando o serviço termina a sua execução.
Classe	Garantir que o invariante da classe se verifica nos tempos estáveis. Garantir que, no fim da execução de cada um dos seus serviços, a respectiva pós-condição se verifica.	Sempre que um dos seus serviços é solicitado, exigir a verificação da respectiva pré-condição.

Tabela 3.1: Programação por contrato (Adaptado de [Meyer 97, página 342]).

pectiva semântica – invariantes da classe, pré-condições e pós-condições dos serviços públicos – por asserções total ou parcialmente[35] executáveis.

Dessa forma não só se torna possível verificar em tempo de execução (ainda que parcialmente) a correcção de cada classe e de cada uma das suas utilizações (dando um novo significado ao mecanismo de excepções como se verá na secção 3.13), como também distribui explícita e claramente as responsabilidades entre as classes e os seus clientes (em oposição à metodologia da programação defensiva [Meyer 97, página 344][Liskov 86]). Assim a classe será responsável por garantir o respectivo invariante nos "tempos" estáveis do seus objectos [Meyer 97, página 364], ou seja, sempre que os objectos podem ser externamente utilizados , assim como garantir as pós-condições dos seus serviços, sendo da responsabilidade dos seus clientes garantir as pré-condições desses serviços (tabela 3.1).

3.12.1 Asserções de classe

As asserções que implementam a semântica dos TDA – invariantes, pré-condições e pós-condições – serão designadas por asserções de classe.

3.12.2 Outras asserções

Embora não tendo a mesma importância do que as asserções de classe, podem-se definir outros tipos de asserções utilizáveis dentro dos algoritmos (preferencialmente estruturados) que implementam os serviços de cada classe. Será o caso das asserções genéricas (instrução `check` em EIFFEL, a *macro* `assert` da biblioteca standard da linguagem C e a instrução `assert` da linguagem JAVA) – aplicáveis em qualquer ponto de um algoritmo – e das asserções associáveis a instruções repetitivas: invariantes e variantes de ciclos (existentes em EIFFEL).

Qualquer que seja a asserção envolvida, a responsabilidade para que ela seja verificada reside sempre no programa envolvido a montante da mesma[36].

[35] É incentivado o uso de comentários nessas asserções, sempre que não seja possível ou conveniente a sua expressão formal [Meyer 97, página 399].

[36] O mesmo se verifica caso a asserção seja concorrente (secção 5.14), embora possa acontecer que parte do programa a montante ainda não tenha sido executado aquando da primeira verificação da asserção.

3.12.3 Asserções e interface de classes

Um aspecto imprescindível para que uma linguagem suporte a programação por contrato é a necessidade de as asserções de classe fazerem parte da interface da classe (ou seja, do TDA). Quer os clientes, quer os herdeiros, de uma classe têm de ser obrigados a cumprir o contrato da classe. Se for permitido que tal não aconteça, então cai por terra a descrição de classes como implementações de TDAs e o polimorfismo de subtipo. Assim, no caso do mecanismo de herança, todos os invariantes das classes ascendentes de uma classe têm de ser herdados (o invariante da nova classe terá de respeitar todos), assim como as pré-condições e pós-condições de cada serviço herdado. Encarando o mecanismo de herança como um meio de sub-contratação [Meyer 97, página 576] (ou seja, classes descendentes têm de pelo menos respeitar os contratos das classes ascendentes) sempre que o polimorfismo de subtipo está envolvido, então as pré-condições podem ser enfraquecidas, e as pós-condições e invariantes podem ser fortalecidos.

Um aspecto muito importante que se deve ter em consideração no uso de asserções é a necessidade de estas serem – tanto quanto possível – aplicativas e não imperativas [Meyer 97, página 351]. Assim todo o cuidado tem de ser tomado para que não se utilizem funções com efeitos colaterais no estado observável do programa em asserções de classe [Meyer 97, página 400].

3.13 Mecanismo de excepções

O factor de qualidade mais importante a se ter em conta num programa é a sua correcção. No entanto, há que também ter em consideração a possibilidade de ocorrerem eventos indesejados em tempo de execução, como sejam falhas no sistema de suporte à execução de programas (por exemplo: falta de memória, de espaço em disco ou do *hardware*), ou no próprio programa pela existência de erros na sua concepção. Para que o programa seja robusto é necessário que essas situações sejam tidas em conta e que exista a possibilidade de lidar com elas de uma forma previsível, e se possível, disciplinada. Essa é a função do mecanismo de excepções em linguagens de programação.

Caso ocorra uma falha no programa uma excepção é gerada (implicitamente pelo sistema de execução do programa, ou explicitamente pelo próprio programa) interrompendo a execução normal desse programa. Essa excepção é propagada pela pilha de execução de serviços do programa, até que seja "apanhada" por código específico para esse efeito, ou até ao fim da pilha, altura em que o programa termina a sua execução. Nessa altura é indicando o ponto do programa onde foi inicialmente gerada a excepção e, se possível, apresentando também o conteúdo da pilha de execução do programa existente nessa altura (já que – na maioria dos casos – o erro deve-se ao programa executado antes do ponto onde a excepção foi gerada).

Se houver a necessidade de garantir a robustez do programa, fazendo com que ele seja tolerante a falhas, o mecanismo de excepções pode servir para suprir essa necessidade sem que haja a necessidade de "contaminar" o algoritmo normal do programa com código específico para essa situação.

Um problema grave de segurança existente na maioria das linguagens com mecanismos de excepções (como acontece em Ada, C++ e Java) consiste na possibilidade de

se "enganar" o programa apanhando uma excepção e deixando que o programa continue a sua execução normal sem resolver o problema que esteve na origem da excepção. O problema aqui assenta na inexistência de uma especificação sobre o que o código que lida com excepções pode ou não fazer. Assim é permitido que esse código "apanhe" uma excepção, escreva uma mensagem de erro, e termine normalmente a execução do serviço onde a excepção foi apanhada sem propagar essa excepção ao restante programa (ainda por cima esta situação é por vezes apresentada como exemplo em livros de apresentação do mecanismo de excepções dessas linguagens). Esta situação interfere negativamente com a relação simples que deve existir entre objectos e TDAs. Uma excepção gerida desta forma pode fazer com que objectos sejam, com ou sem intenção, utilizados fora dos seus tempos estáveis, ou seja para os quais os axiomas dos TDAs possam não fazer sentido.

O que deve então ser permitido no código que apanha e lida com excepções? Meyer [Meyer 97, página 417] defende que – na execução de um serviço – apenas é aceitável uma de duas acções:

1. Tentar corrigir a causa da excepção e voltar a executar o serviço (*retrying*);
2. Repor, um estado estável no objecto, e reportar a falha (propagando a excepção) ao cliente do serviço (*failure*).

Dessa forma, deixa de ser possível permitir que o programa continue a sua execução normal sem que a causa da excepção não esteja corrigida. O mecanismo de excepções em Eiffel baseia-se neste comportamento, sendo por isso designado por mecanismo disciplinado de excepções.

Outro aspecto essencial do mecanismo de excepções existente em Eiffel é a sua relação com as asserções. Assim sempre que uma asserção não é verificada é gerada uma excepção, dando assim total coerência e simplicidade à implementação dos TDAs em Eiffel[37]. Temos assim um aproveitamento sinérgico de todos esses diferentes mecanismos simplificando e dando consistência à linguagem (esta integração elegante será com certeza uma das razões mais fortes pela qual esta linguagem cativa muitos dos programadores que a ela são expostos).

É importante referir que o mecanismo de excepções serve para lidar com falhas no sistema de suporte à execução de programas e erros em programas. Não serve para situações normais e previsíveis que devem fazer parte da especificação de programas. A utilização deste mecanismo para essas situações mais não representa do que a adopção encapotada de uma instrução de "saltos" com todos os problemas de complexidade que lhe estão associados.

3.14 Polimorfismo *ad-doc*: sobrecarga de serviços

Algumas linguagens com um sistema de tipos estático (C++, Java) permitem que uma classe possa ter vários serviços com o mesmo nome, desde que as respectivas assinaturas sejam estaticamente diferentes. O serviço a ser executado é decidido em

[37] A lista completa de situações que geram excepções em Eiffel pode ser encontrada em [Meyer 97, página 413].

tempo de compilação consoante as respectivas assinaturas. Este tipo de polimorfismo é designado por *ad-doc*[38] [Cardelli 85].

Este mecanismo, parecendo ser útil em alguns casos particulares, gera problemas complicados de ambiguidade e segurança na linguagem. A ambiguidade resulta do facto de o nome de um serviço de uma classe poder já não ser suficiente para o localizar. A situação complica-se ainda mais caso a estrutura de classes ascendentes dessa classe seja complexa.

Uma interferência insegura inevitável ocorre com o mecanismo de polimorfismo de subtipo. Essa situação é exemplificada com o seguinte programa:

```
class A
  ...
end;

class B
inherit A
  ...
end;

class C
feature
  p(a: A) is ... end;
  p(b: B) is ... end; -- invalid Eiffel!
end;
```

```
...

  local
    a: A;
    b: B;
    c: C;
  do
    a := b;
    c.p(a); -- (1)
    c.p(b); -- (2)
  end;
```

Assim, embora as invocações em `(1)` e `(2)` sejam em tempo de execução iguais (ambas passam um objecto do tipo `B`) elas são tratadas diferentemente pelo programa (não sendo assim orientado por objectos, mas sim pelo tipo estático da entidade que os manipula).

Caso a linguagem tenha herança múltipla, teremos outra fonte de interferências potencialmente inseguras deste mecanismo. Assim, quer o programador o deseje explicitamente, ou tão só por mera distracção, passa a ser permitido herdar serviços com o mesmo nome desde que tenham assinaturas diferentes (por exemplo, os dois serviços `p` da classe `C` do exemplo anterior, poderiam vir, de uma forma não intencional, de duas classe ascendentes distintas).

Por todas estas razões, e mesmo tendo em consideração as poucos situações onde este mecanismo parece ter alguma utilidade, parece-nos muito discutível a sua adopção.

3.15 Gestão de memória

Existem linguagens que delegam no programador a responsabilidade de gerir a memória utilizada pelo programa (C++) e outras que assumem essa responsabilidade automatizando essa gestão (Eiffel, Java).

Por um lado a gestão "manual" da memória permite afinar esse processo garantindo que o sistema de execução não está a gastar recursos (em particular ciclos de execução da unidade de processamento central) em alturas menos próprias. Por outro, essa gestão é extremamente sensível a erros e omissões por parte dos programadores,

[38] Cardelli identifica outra forma de polimorfismo *ad-hoc*, que aqui não será abordada, associado à coerção de tipos.

gerando consequências catastróficas para o programa em execução (quando por exemplo, as mesmas porções de memória estão a ser utilizadas "simultaneamente" para fins diferentes), ou perdas progressivas de memória livre do sistema de execução. Linguagens como o C++ que permitem manipulações complexas (como o uso de operações aritméticas) com apontadores de memória agravam ainda mais estes problemas, podendo tornar difícil a compreensão dos programas e a detecção e correcção de erros subtis de gestão de memória no programa.

A gestão automática de memória recorrendo a colectores de "lixo" evitam estes problemas sérios de correcção dos programas e de segurança da própria linguagem, simplificando, simultaneamente, o trabalho ao programador. As objecções que se podem levantar em aplicações específicas, por exemplo, de garantias de tempos de execução em tempo real, podem ser atenuadas caso seja possível ajustar os parâmetros do colector de lixo em tempo de execução (permitindo, por exemplo, que ele seja temporariamente desactivado).

3.16 Serviços de classe

Linguagens como o C++ e o JAVA permitem a definição de serviços de classe que são partilhados por todas as instâncias directas ou indirectas da classe onde são declarados (serviços tipo **static**). Este tipo de serviços pode ser invocado independentemente de uma entidade com tipo estar associada a um objecto, o que em certas situações pode ser útil.

Por exemplo caso se queira construir uma classe `DATA`, constituída pelos serviços dia, mês e ano, e caso se queira garantir que as suas instâncias representam sempre uma data válida – ou seja, essa condição será um dos invariantes da classe – então é útil a existência de um serviço de classe que sirva para validar datas, permitindo assim essa validação a eventuais clientes, sem ser necessário implementar esse serviço fora da classe.

No âmbito deste trabalho é necessário incluir uma análise deste mecanismo porque ele – pela sua própria definição – interfere directamente com alguns mecanismos de concorrência.

3.17 Serviços de execução única

A linguagem EIFFEL introduz um outro tipo de serviços: serviços de execução única.

Originalmente estes serviços garantiam uma execução única para todas as instâncias (directa ou indirectas) da classe onde eram definidos, sendo por isso muito úteis para inicialização (procedimentos) e partilha de objectos (funções). Evoluções mais recentes da linguagem [ECMA-367 05] permitem a definição de contextos de execução única diferenciados recorrendo a chaves textuais, estando também a ser pensada a possibilidade de futuramente permitir outros contextos como sejam o mesmo processador (como alternativa de ele se aplicar a todo o programa), somente para o objecto, para a classe, e para chaves livres[39].

[39] A modificação feita pelo autor do compilador SMALLEIFFEL (apêndice D) implementa todas essas variantes.

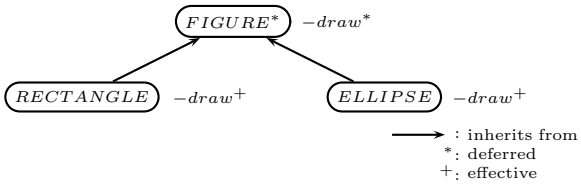

Figura 3.4: Exemplo serviço abstracto.

3.17.1 Comparando com os serviços de classe

É interessante a comparação entre os serviços de execução única e os serviços de classe. Ambos são um mecanismo de partilhar serviços para além do próprio objecto. No entanto o alcance dessa partilha, e a semântica na sua execução é significativamente diferente.

Enquanto que nos serviços de classe o alcance da partilha se aplica sempre a todas as instâncias de classes descendentes[40] da classe onde esses serviços estão declarados; já no caso dos serviços de execução única esse alcance pode ser adaptado a várias situações (para a classe, para o objecto, para todos os processadores, ou apenas para um).

Na semântica de execução os serviços de classe são executados sempre que requerido, enquanto que os serviços de execução única apenas o são uma vez, sendo que nas restantes invocações ou nada fazem – se forem procedimentos – ou – no caso das funções – simplesmente devolvem o valor retornado na primeira invocação. São assim uma forma bastante elegante quer de inicialização partilhada de recursos, quer de partilha de objectos.

Os atributos de classe são também uma forma de partilha de objectos. No entanto, diferem das funções de execução única por poderem ter efeitos colaterais. Nesta perspectiva as funções de execução única têm uma aproximação mais funcional, em contraste com a aproximação mais imperativa dos atributos de classe.

Uma vez que as funções de execução única servem para partilhar objectos, isso quer dizer que os serviços dos objectos por essa forma partilhados se comportam – caso o alcance se aplique a todas as instâncias da classe – como se fossem serviços de classe.

Estas diferentes propriedades, como seria de esperar, terão consequências bastante diferentes na sua integração com mecanismos concorrentes como se verá no capítulo 5.

3.18 Serviços "abstractos"

Um mecanismo muito útil na concepção e construção de programas orientados por objectos, é a possibilidade de se poder declarar em classes (não instanciáveis) apenas a interface de alguns dos seus serviços, relegando possíveis implementações para classes descendentes (serviços abstractos puros em C++ e **deferred** em Eiffel).

A existência deste mecanismo permite maximizar as utilizações do polimorfismo de subtipo. A sua utilidade é bem demonstrada pela exemplo apresentado na figura 3.4.

[40] Incluindo a própria.

A classe `FIGURE` não tem qualquer possibilidade de dar uma implementação que faça sentido para o serviço `draw`, pelo que a possibilidade de definir serviços sem implementação resolve esse problema (para além de evitar a instanciação de objectos como instâncias directas dessa classe).

Temos assim que os serviços abstractos permitem a construção de classes sem a necessidade de (eventualmente) lhes associar uma representação interna, já que tal situação, em muitos casos, seria uma sobre-especificação do TDA da classe.

3.19 Juntando tudo: interferências entre mecanismos

O número de diferentes mecanismos existentes em linguagens orientadas por objectos excede largamente os que aqui foram apresentados. No entanto, neste trabalho, optou-se por apresentar aqueles que são considerados essenciais, e os que se julga ser mais importantes (geralmente pela positiva, embora aqui e ali também pela negativa, como aconteceu com a sobrecarga de serviços). Outro critério que tivemos em consideração nesta apresentação, foi incluir mecanismos que pela sua própria definição interfiram com a inclusão de mecanismos de concorrência nessas linguagens (como se verá no capítulo 5).

Nesta secção iremos completar a apresentação das linguagens orientadas por objectos resumindo algumas das possíveis interferências, inseguras ou sinérgicas, no uso conjunto desses mecanismos.

Como se analisou no capítulo anterior, a compreensão de como os mecanismos de uma linguagem podem interferir negativa ou sinergicamente entre si, é um aspecto absolutamente essencial para se aferir a qualidade da linguagem como um todo. A qualidade da linguagem será assim tanto maior quanto mais garantir a inexistência de interferências inseguras entre mecanismos, e quanto mais proveito tirar de interferências sinérgicas com sentido entre os mesmos.

A tabela 3.3 sintetiza algumas das interferências inseguras mais importantes que podem ocorrer entre alguns dos mecanismos das linguagens orientados por objectos, assim como soluções possíveis para essas situações.

De uma forma similar, a tabela 3.4 apresenta algumas propriedades sinérgicas importantes para a programação por objectos.

Cada letra apresentada na primeira coluna dessas tabelas corresponde a um mecanismo em particular, sendo estes apresentados na tabela 3.2.

Por fim, na tabela 3.5 apresenta-se uma síntese das características de algumas das linguagens orientadas por objectos mais importantes.

Neste trabalho, talvez com a excepção da linguagem Eiffel[41], optámos por fazer uma abordagem mais orientada às propriedades e mecanismos da programação por objectos (quer individualmente, quer nas suas propriedades conjuntas), e não a uma análise detalhada de cada uma das linguagens orientadas por objectos. Uma tal apresentação detalhada (para além da tabela apresentada) não traria em nossa opinião nenhuma mais-valia para este trabalho, podendo mesmo dificultar a compreensão do trabalho realizado.

[41] Que serviu de base para a realização dos mecanismos estudados e propostos.

A:	Instruções estruturadas puras	**J:**	Mecanismo de Excepções
B:	Encapsulamento de informação	**K:**	Sobrecarga de serviços
C:	Herança simples	**L:**	Serviços de classe
D:	Polimorfismo subtipo e encaminhamento dinâmico simples	**M:**	Serviços de execução única
E:	Objectos como Tipos de Dados Abstractos	**N:**	Serviços abstractos
F:	Polimorfismo paramétrico	**O:**	Modificação externa directa de atributos
G:	Herança múltipla	**P:**	Separação entre comandos e consultas
H:	Com programação por contrato	**−:**	Interferência negativa (insegura)
I:	Sem programação por contrato	**+:**	Interferência positiva (sinérgica)

Tabela 3.2: Legenda de mecanismos.

–	**Descrição:**	**Linguagens:**	**Soluções:**	**Refs.:**
A – J	instruções estruturada puras podem ser interrompidas por excepções, podendo o programa continuar sem garantir a pós-condição que lhes está implicitamente associada	Ada95, C++, Java	adoptar o mecanismo de excepções disciplinadas	(página 34), (página 34)
B – C/D	classe descendente podendo ter um encapsulamento mais restritivo	Eiffel	análise global do programa, *CAT-Calls*	(página 21), (página 24)
B/E – F	TDA do parâmetro de tipo pode esconder serviços requeridos pela classe paramétrica	C++	mecanismo de polimorfismo paramétrico restringido	(página 29), (página 29)
B/E – H	pré-condições utilizando serviços não exportados	Eiffel	impedir estaticamente essa situação na fase de compilação	[Meyer 97, página 357]
C – D	definição covariante do tipo de atributos ou dos argumentos de serviços	Eiffel	análise global do programa; proibir *catcalls* polimórficos; encaminhamento dinâmico múltiplo	(página 24)
C/D – I	a linguagem não obriga a que as classes respeitem o TDAs das classes ascendentes	C++, Java, Ada95	análise global do programa; proibir *catcalls* polimórficos; encaminhamento dinâmico múltiplo	(página 24)
C/G – K	sobrecarga não intencional de serviços herdados	C++, Java	eliminar o mecanismo K	(página 31)
D – K	ambiguidade na selecção dos serviços a serem executados	C++	eliminar o mecanismo K	(página 35)
E – J	mecanismos de excepções não disciplinados permitem que objectos sejam utilizados fora dos seus tempos estáveis	Ada95, C++, Java	impor o mecanismo de excepções disciplinadas	(página 34)
E – O	a classe deixa de ser a única responsável por garantir o seu invariante	C++, Java	eliminar a propriedade O	(página 21)
G – G	colisão de nomes	C++	mecanismo de mudança de nomes	(página 31)
G – G	na presença de herança repetida que serviços da classe ascendente herdada várias vezes devem ser duplicados, e quais devem ser partilhados	C++	mecanismo de mudança de nomes	(página 31)
H – H	uso de funções com efeitos colaterais no estado observável do programa em asserções	Eiffel	permitir apenas o uso de funções puras em asserções	(página 33)
I – J	não há a garantia de que erros de correcção no programa gerem excepções	C++, Java, Ada95		(página 34)
J – J	não propagar excepções cuja causa não tenha sido resolvida	Ada95, C++, Java	adoptar o mecanismo de excepções disciplinadas	(página 34), (página 34)
J – J	utilização de excepções para o algoritmo normal do programa	Todas	adoptar o mecanismo de excepções disciplinadas, e restringir o uso de asserções apenas para aferir a correcção do programa	(página 34), [Meyer 97, página 346]

Tabela 3.3: Algumas interferências inseguras entre mecanismos.

+	**Descrição:**	**Linguagens:**	**Refs.:**
A + P	detecção de funções puras	Eiffel	(página 19)
B + E	os TDAs definem o encapsulamento desejável para cada objecto	Todas	(página 27)
B + H	o encapsulamento com asserções de classe implementam o TDA dessa classe	Eiffel	(página 32)
C/D + H	herança de contratos: sub-contratação	Eiffel	(página 33)
D + F	polimorfismo paramétrico restringido	Eiffel	(página 29)
E + N	os serviços abstractos permitem construir classes sem implementação, ou com uma implementação parcial, para o seu TDA	Eiffel	(página 38)
H + J	uma vez que as asserções servem para aferir a correcção de programas, as excepções são a resposta adequada sempre que há um incumprimento de contratos	Eiffel	(página 34)
H + P	as asserções só devem utilizar serviços do tipo consulta (sem efeitos colaterais)	Eiffel	(página 34)

Tabela 3.4: Algumas interferências sinérgicas entre mecanismos.

Linguagem	Origem	Características	Referências
SIMULA	1967[a]	Primeira linguagem com mecanismos da programação por objectos. Sem encapsulamento de informação. Serviços abstractos. Herança simples (inicialmente designada por concatenação). Polimorfismo subtipo. Encaminhamento dinâmico (mas não por omissão). Gestão automática de memória.	[Dahl 68]
Smalltalk	1972	Primeira vez onde é utilizada a denominação orientado por objectos. Linguagem orientada por objectos pura. Sistema de tipos dinâmico. Classes podem ser manipuladas como objectos (meta-classes). O encapsulamento de informação é predefinido para esconder todos os atributos e tornar públicos todos os métodos. Herança simples. Encaminhamento dinâmico. Serviços de classe. Gestão automática de memória.	[Goldberg 89]
C++	1983	Linguagem híbrida que projectada como uma extensão da linguagem C com mecanismos orientados por objectos. Sistema de tipos estático. Encapsulamento de informação. Herança múltipla. Polimorfismo subtipo. Encaminhamento dinâmico (mas não por omissão). Polimorfismo paramétrico. Sobrecarga de métodos e operadores. Serviços de classe. Mecanismo de excepções. Gestão manual de memória.	[Stroustrup 85, Stroustrup 97]
Eiffel	1986	Linguagem orientada por objectos pura. Sistema de tipos estático. Encapsulamento de informação ajustável por cliente. Polimorfismo subtipo. Encaminhamento dinâmico. Herança múltipla. Suporte para programação por contrato. Polimorfismo paramétrico (restringido). Serviços de execução única. Mecanismo disciplinado de excepções. Gestão automática de memória.	[Meyer 88b, Meyer 92, Meyer 97]
Ada95	1995[b]	Linguagem híbrida. Aproximação incompleta à programação por objectos (há uma separação sintáctica entre dados - *tagged record types* - e funções/procedimentos). Sistema de tipos estático. Encapsulamento de informação. Herança simples. Encaminhamento dinâmico (mas não por omissão). Polimorfismo paramétrico. Mecanismo de excepções. Suporte para programação concorrente.	[Ada95 95]
Java	1995	Sistema de tipos estático. Encapsulamento de informação. Herança simples. Encaminhamento dinâmico. Interfaces (com herança múltipla de outras interfaces). Sobrecarga de serviços. Mecanismo de excepções. Gestão automática de memória. Suporte para programação concorrente.	[Gosling 96, Gosling 05]

[a] Existiu uma versão anterior de 1964, conhecida por SIMULA 1.

[b] A primeira versão de Ada é de 1979, mas apenas em 1995 é que a linguagem se aproximou da orientação por objectos.

Tabela 3.5: Descrição de algumas linguagens orientadas por objectos.

Capítulo 4

Programação Concorrente Procedimental

Neste capítulo faz-se uma descrição da programação concorrente procedimental apresentando os seus problemas e desafios assim como soluções mais comuns para os mesmos. Foram excluídas propositadamente as aproximações orientadas por objectos à programação concorrente que serão tratadas no próximo capítulo.

4.1 Conceitos básicos

Um programa concorrente distingue-se de um sequencial por poder ser composto por mais do que um "sub-programa" com execução autónoma. Em geral, esses "sub-programas" apesar de terem uma execução autónoma, cooperam entre si para que o programa no seu todo atinja um ou vários objectivos comuns (razão pela qual fará sentido chamar-lhe um programa, e não um conjunto de programas independentes).

Por convenção iremos designar as entidades que executam os "sub-programas" por "processadores"[1], sendo estes definidos da seguinte forma[2]:

Processador

Um processador é uma unidade de processamento autónoma capaz de suportar a execução sequencial de instruções.

Iremos também designar esses "sub-programas" por programas de cada processador.

É importante tornar claro que um programa concorrente não implica necessariamente a execução simultânea (no tempo) de processadores. Essa execução pode ser, por exemplo, ciclicamente alternada ao longo do tempo tal como acontece nos sistemas

[1] Muitos autores (por exemplo [Andrews 83]) utilizam a denominação "processo" para o mesmo efeito. No entanto, optou-se por mesmo assim utilizar uma denominação distinta já que essa designação é frequentemente utilizada para uma concretização em particular de processadores em sistemas operativos com escalonamento preemptivo. Desta forma, espera-se evitar possíveis confusões com essa concretização em particular.

[2] Esta definição é similar à utilizada por Meyer [Meyer 97, página 964] para a extensão concorrente SCOOP proposta para a linguagem EIFFEL.

operativos com escalonamento preemptivo de processos em computadores com uma única unidade de processamento. No caso particular em que a execução é garantidamente simultânea (como pode acontecer por exemplo em arquitecturas *SMP*), é usual designar a programação concorrente como sendo programação paralela.

4.1.1 Abordagem explícita à concorrência

Podemos definir duas abordagens possíveis para se construírem programas concorrentes: uma explícita e outra implícita. Na primeira, é da responsabilidade do programador o uso explícito de abstracções concorrentes apropriadas para os programas de cada processador, sendo visível para este quais as partes do programa que são executadas em concorrência. Na segunda abordagem, a responsabilidade de particionar um programa em "sub-programas" concorrentes cabe exclusivamente ao sistema de compilação e de execução. Para que tal objectivo seja atingível torna-se necessário o uso de linguagens de programação apropriadas que não imponham dependências sequenciais excessivas aos programas, como é o caso de linguagens declarativas. A adopção desta última abordagem em linguagens imperativas é bastante mais complexa já que estas linguagens tendem a impor uma sequenciação rígida nos algoritmos, dificultando a sua paralelização (neste aspecto podemos dizer que as linguagens imperativas são mais susceptíveis a sobre-especificar a construção de algoritmos).

Como é óbvio existe também a possibilidade de se fazerem abordagens conjuntas à programação concorrente, recorrendo simultaneamente a mecanismos explícitos de concorrência e a algoritmos de paralelização automática em tempo de compilação e execução. No entanto, neste trabalho iremos concentrar a nossa atenção apenas nas abordagens explícitas à concorrência.

4.1.2 Sistemas de programação concorrente

As aproximações à programação concorrente podem ser baseadas em bibliotecas de *software* (é o caso da biblioteca POSIX *threads* para a linguagem C[3] [Butenhof 97]); em linguagens concorrentes (Concurrent Pascal [BH 75]); ou numa mistura de ambas (Java). Iremos designar indistintamente por sistema de programação concorrente uma qualquer destas aproximações.

4.1.3 Processadores abstractos

Em sistemas de programação concorrentes é usual a associação de processadores a suportes de execução de programas específicos, como sejam processos em sistemas operativos, ou a diferentes *threads* dentro de um único processo. É o caso, por exemplo, da linguagem Java cujos mecanismos de concorrência estão estaticamente ligados a *threads*. No entanto, a larga maioria das propriedades dos programas concorrentes não dependem de suportes específicos de execução de cada processador, pelo que essa aproximação de associar rigidamente cada processador a um único suporte de execução é, em muitos casos, claramente uma sobre-especificação (como já foi referido, esta foi uma das razões para o uso do termo "processador" em vez de "processo"). Será preferível

[3] O sistema de compilação tem, no entanto, de ser informado dessa situação.

permitir a eventual definição de diferentes suportes de execução para cada processador, como por exemplo: processos, *threads*, um conjunto de processos envolvendo um grupo de computadores em rede, ou recorrendo a sistemas de suporte à programação paralela e distribuída como o *PVM* [Geist 94] ou o *MPI* [Forum 94].

O sistema será classificado como tendo *processamento heterogéneo* se permitir a associação de diferentes dispositivos de processamento a processadores, caso contrário será designado como um sistema de *processamento homogéneo*.

O processamento heterogéneo é uma propriedade desejável para sistemas concorrentes já que reforça a separação entre programas e dispositivos de suporte à execução, tornando-os mais facilmente adaptáveis a novos contextos de execução. É, no entanto, importante referir que existem certos casos particulares de programação concorrente, como a programação em tempo real ou em sistemas embutidos[4], onde podem ser colocadas restrições fortes a associações heterogéneas de processadores, de forma a que os programas cumpram os objectivos para os quais foram construídos.

4.1.4 Escalonamento de processadores

Quando existem mais processadores do que dispositivos de processamento, ou quando há uma competição de vários processadores por um recurso partilhado torna-se necessário seleccionar quais os processadores a executar. À estratégia usada para essa selecção chama-se escalonamento de processadores.

Em geral, três factores estão envolvidos nesse escalonamento [Ruschitzka 77]:

- O modo de decisão;
- A função de prioridade;
- A regra de arbitragem.

O modo de decisão caracteriza os instantes de tempo nos quais é decidido o escalonamento de processadores (por exemplo, em sistemas operativos de partilha de tempo preemptivo, estes instantes ocorrem com uma frequência constante). A função de prioridade consiste no algoritmo de ordenação de processadores. E por fim, a regra de arbitragem é a estratégia utilizada para escolha entre processadores de igual prioridade.

A escolha do escalonamento pode afectar a segurança dos programas concorrentes, uma vez que ela pode prevenir alguns problemas de *deadlock*, ou – quando é utilizado um algoritmo extremamente "injusto" – pode colocar problemas de *liveness*[5] (ver secção 4.2.2) tais como nunca escolher para execução algum processador (*starvation*).

Este trabalho não irá abordar os problemas associados ao escalonamento de processadores. Assumir-se-á que o sistema de suporte à concorrência garante alguma equidade no acesso à execução para todos os processadores existentes.

4.1.5 Programação em tempo-real

Uma área muito importante da programação concorrente que não irá ser abordada neste trabalho é a da programação em tempo-real. Neste tipo de programas é essencial

[4] *embedded systems*.

[5] Não encontrei uma tradução aceitável para este termo.

garantir não apenas a correcção (e robustez) lógica dos programas, mas também a sua correcção temporal. A correcção temporal verifica-se quando é garantido que os vários componentes do programa terminam a sua execução dentro de limites temporais impostos na especificação do programa[6].

Muita da evolução da programação tem assentado na abstracção do tempo na execução de programas (reduzindo-o tão só a uma imposição de causalidade lógica entre as várias acções de um programa) pelo que a programação em tempo-real obriga, de alguma forma, a reformular os programas por forma a que o tempo de execução volte a ser um dos seus aspectos essenciais.

Geralmente as abordagens a este tipo de programação assentam no uso de bibliotecas e sistemas de suporte à execução específicos para tempo-real (sistemas operativos de tempo-real). Já o uso de mecanismos de linguagens específicos para programação em tempo-real é muito raro, muito embora à partida nos pareça que que os mesmos poderiam facilitar essa programação (tornando-a mais abstracta, logo mais simples). Esta será uma das áreas nas quais se espera desenvolver futuramente trabalho.

4.2 Correcção de programas concorrentes

Lamport [Lamport 83] define dois grupos de propriedades essenciais a ser verificadas em programas concorrentes:

- segurança;
- *liveness*.

4.2.1 Segurança

Os programas concorrentes podem criar problemas de segurança (ver definição de segurança na secção 2.2.4) muito complexos e por vezes de difícil detecção. Estes problemas estão sempre ligados a sincronizações incorrectas entre processadores (a sincronização de processadores é apresentada à frente na secção 4.6).

Este tipo de erros é sem dúvida o problema de correcção mais sério colocado pela programação concorrente, já que podem depender do tempo de execução relativo de cada processador (que em geral não é de todo previsível e controlável), sendo em muitos casos difíceis de reproduzir e detectar.

Os erros por competição dessincronizada[7] são o mais simples deste tipo de problemas. Esses erros ocorrem sempre que não há uma sincronização adequada de um recurso partilhado e existem vários processadores a competir entre si no acesso a esse recurso. Esta situação pode fazer com que nenhum dos processadores faça correctamente aquilo que pretende, deixando o recurso partilhado num estado inconsistente. Uma possível solução para este problema é proteger o acesso a esse recurso dentro de uma região crítica, utilizando por exemplo semáforos [Dijkstra 68a].

[6] O que não quer dizer que os programas têm de executar o mais eficientemente possível, mas tão só apenas o suficiente para garantir a especificação temporal.

[7] *race conditions*.

A gravidade dos problemas de segurança em programas concorrentes justifica que se procurem mecanismos de linguagens que garantam a inexistência desses problemas (aproximação axiomática ao sincronismo). Quando, pelo contrário, se passa a responsabilidade de uma sincronização correcta para as mãos dos programadores (aproximação operacional ao sincronismo), como acontece na larga maioria dos sistemas de programação concorrente actualmente utilizados, existe sempre o risco de insegurança nos programas.

4.2.2 Propriedades de *liveness*

Lamport [Lamport 83] apresenta estas propriedades como sendo aquelas que descrevem o que o programa tem de fazer. Ou seja, essas propriedades, a verificarem-se, garantem que os programas atingem determinados fins.

Em programas concorrentes existem várias situações que podem impedir a verificação dessas propriedades.

Deadlocks

Os *deadlocks*, que Dijkstra designou originalmente por "abraço mortal entre processadores" [Dijkstra 68a], são situações em que processadores esperam eternamente por recursos reservados por outros. Para que esta situação ocorra, é necessário que se verifiquem quatro condições [Coffman 71]:

1. Exclusão mútua (acesso exclusivo a recursos);
2. Reserva e espera (espera pelo acesso a um recurso enquanto mantém reservado para si próprio pelo menos um outro recurso);
3. Não preempção (uma vez um recurso reservado por um processador, só o próprio é que o pode libertar);
4. Espera circular.

É suficiente que uma destas condições não se verifique para garantir a inexistência de *deadlocks*.

Existem três estratégias para atacar este problema [Coffman 71]:

1. Prevenção estática;
2. Prevenção dinâmica;
3. Detecção.

A prevenção estática garante a inexistência de *deadlocks* fazendo com que em tempo de compilação (estaticamente) pelo menos uma das quatro condições não se verifique. Por exemplo, permitindo que um processador reserve no máximo um recurso de cada vez (preempção permitida), ou exigindo que os processadores reservem todos os recursos que necessitam de uma só vez (reserva e espera negada), ou se é imposta uma reserva ordenada dos recursos (espera circular impossível); então os *deadlocks* não podem ocorrer. Chama-se no entanto a atenção de que tem de haver algum cuidado na

utilização destas técnicas de prevenção já que elas tendem a ser muito penalizadoras para o desempenho global do programa.

Outra estratégia segura consiste em utilizar técnicas de prevenção dinâmica de *deadlocks*. Se existir informação acerca da actual e futura possível ocupação de recursos, então esse conhecimento pode ser utilizado para evitar esperas circulares (como, por exemplo, o *algoritmo do banqueiro* [Dijkstra 68a, Habermann 69]).

A terceira possibilidade consiste em ter algoritmos de detecção de *deadlocks*, e estratégias de reparação (que podem reutilizar o próprio mecanismo de excepções da linguagem).

Apenas as primeiras duas estratégias são garantidamente seguras, já que não afectam a execução normal dos processadores, devendo assim ser as principais a considerar na construção de linguagens seguras.

Ao contrário da exclusão mútua – que é um problema local com soluções locais – os *deadlocks* surgem como resultado de uma interferência global no programa entre processadores. Esta característica faz com que este problema seja muito mais difícil de lidar.

Outros problemas

A ocorrência de *deadlocks* é sem dúvida o problema mais frequente na garantia de *liveness* de programas concorrentes. Não é no entanto o único. Podem também existir problemas de *livelocks*, que, tal como os *deadlocks* impedem eternamente (a não serem resolvidos, é claro) a progressão do programa (ou parte dele), mas com a diferença de não ser por bloqueamento passivo dos processadores mas sim um bloqueamento activo em que estes estão num processo de espera ocupada[8] uns pelos outros.

Outro problema possível é a suspensão eterna (*starvation*) de um processador (ou de vários) tão só porque o sistema de escalonamento de processadores nunca o selecciona para execução.

4.3 Requisitos essenciais

Neste trabalho estamos interessados no estudo de mecanismos de linguagens para programação concorrente orientada por objectos com processadores abstractos sem requisitos de tempo-real. Um primeiro passo nesse sentido passará por identificar claramente quais os requisitos essenciais colocados na programação concorrente procedimental. Esses requisitos distribuem-se em três grupos [Andrews 83]:

- Execução concorrente de processadores;
- Comunicação entre processadores;
- Sincronização entre processadores.

[8] *busy waiting*.

4.4 Execução concorrente de processadores

Os sistemas de programação concorrentes têm de ter mecanismos apropriados para iniciar, suportar e terminar a execução de processadores. Este comportamento básico pode ser obtido directamente através de mecanismos específicos de linguagens de programação, ou indirectamente recorrendo a bibliotecas apropriadas de *software*. A primeira aproximação será a escolha natural a ser feita em linguagens concorrentes pois permite que o sistema de compilação conheça os pontos do programa onde são criados novos processadores. Dessa forma, tornam-se explícitos os sub-programas associados aos processadores, melhorando o conhecimento do sistema de compilação sobre o programa concorrente. A segunda aproximação justifica-se quando se pretende introduzir concorrência em linguagens sequenciais sem as modificar[9]. Nesta situação é importante referir-se que, apesar de a linguagem base poder não ser afectada, o mesmo não acontece ao sistema de compilação. Este, por forma a gerar programas concorrentes a funcionar devidamente, em geral terá sempre de saber se o programa a ser compilado é ou não concorrente. Neste trabalho iremos debruçar-nos exclusivamente sobre a primeira aproximação.

4.4.1 Instrução estruturada de execução concorrente

Uma possibilidade para se expressarem processadores consiste no uso de instruções de execução concorrente. Dijkstra [Dijkstra 68b, página 12] propôs uma dessas instruções estruturadas de execução concorrente como uma extensão à linguagem ALGOL 60[10]:

```
begin
   S1;
   parbegin
     S2;
     S3;
     S4;
   parend;
   S5
 end
```

Sendo `S1`, `S2`, `S3`, `S4` e `S5` blocos de instruções da linguagem, o comportamento do programa será executar `S1` seguindo-se a execução concorrente de `S2`, `S3` e `S4` e por fim, e só após esses três blocos terminarem a sua execução, é que será executado `S5`. A figura 4.1 mostra o grafo de execução deste programa.

Esta instrução, muito embora tenha a propriedade muito importante de ser estruturada pura (página 16), limita a expressividade da linguagem já que apenas permite a construção de programas concorrentes em que grupos de processadores são sempre criados e destruídos em conjunto. Outra limitação desta instrução é que apenas permite expressar um número estaticamente predefinido de processadores (no exemplo dado, três).

[9] É o caso do POSIX-THREADS para a linguagem C.

[10] Um nome mais apropriado para este tipo de instruções será **cobegin-coend** [Andrews 83, página 8].

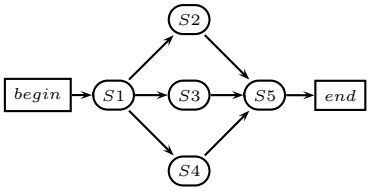

Figura 4.1: Exemplo de instrução estruturada de execução concorrente.

4.4.2 Instruções de bifurcação e junção de processadores

Uma alternativa mais genérica para criar novos processadores assenta na instrução **fork** [Conway 63, Dennis 66]. Esta instrução permite criar um novo processador associado à execução (concorrente) de um procedimento. Esta instrução é complementada com a instrução **join** utilizada para fazer com que um processador espere até que um procedimento concorrente termine a sua execução. O exemplo atrás apresentado implementado com estas instruções teria o seguinte aspecto (assumindo que `S2`, `S3` e `S4` seriam invocações a procedimentos):

```
begin
   S1;
   fork S2;
   fork S3;
   fork S4;
   join S2;
   join S3;
   join S4;
   S5
end
```

Com este grupo de instruções, ao contrário da instrução anterior, é já possível expressar qualquer grafo de execução concorrente de programas assim como criar um número ilimitado de processadores em tempo de execução.

4.4.3 Associação estática de processadores a procedimentos

Uma outra possibilidade consiste em associar estaticamente processadores a procedimentos. Nesta situação a execução desses procedimentos será, por definição, concorrente com o programa que os invoca.

Caso possa haver apenas uma instância de cada um desses procedimentos, o número de processadores será estaticamente imposto. Se, como alternativa, a esses procedimentos puder estar associado um tipo, então teremos a possibilidade de existirem múltiplas instâncias de cada um desses procedimentos, permitindo um número dinamicamente variável de processadores.

4.5 Comunicação entre processadores

Existem essencialmente dois modelos (abstractos) de comunicação entre processadores [Andrews 83]:

- Envio de mensagens (comunicação directa);

- Partilha de memória (comunicação indirecta).

No modelo de comunicação por envio de mensagens, os processadores comunicam directamente utilizando um qualquer canal de comunicação entre eles. Existe assim um processador emissor (cliente) e outro receptor (servidor) havendo a possibilidade de a comunicação se processar, após o envio da mensagem pelo primeiro processador, apenas quando o processador receptor estiver disponível (e disposto) a que tal aconteça. Esta forma de comunicação está bem adaptada a processadores pouco dependentes uns dos outros (fracamente ligados), como acontece em sistemas distribuídos, ou em topologias cliente-servidor.

A comunicação por partilha de memória é um mecanismo indirecto, no qual a comunicação é feita utilizando uma entidade partilhada que pode ser modificada e observada. Este modelo de comunicação está bem adaptado para situações em que os processadores necessitam frequentemente de partilhar informação mutável (fortemente ligados).

Como é referido em [Lauer 78], qualquer um dos dois modelos de comunicação pode ser simulado com o outro, pelo que se pode argumentar que em princípio um sistema de programação concorrente apenas necessitaria de um deles. No entanto, essa conversão representa quase sempre uma perda não só de eficiência como também e principalmente de expressividade, já que ambos representam abstracções de comunicação diferentes. É assim defensável a adopção de ambos os modelos em linguagens concorrentes.

4.5.1 Comunicação síncrona e assíncrona

Um mecanismo de comunicação é definido como síncrono em relação a um processador no caso dessa comunicação, do ponto de vista desse processador, só terminar quando for realizada com sucesso. Nesta situação o processador pode ser obrigado a um período de espera (bloqueamento), até que a comunicação se efectue[11]. No caso em que não é requerido o processamento integral da comunicação antes de o processador poder prosseguir com o respectivo algoritmo, a comunicação diz-se assíncrona. A comunicação pode ser também uma combinação de ambos os casos, quando uma parte da comunicação é síncrona e outra assíncrona. É o caso, por exemplo, do emissor ser síncrono com a colocação da mensagem na fila de mensagens pendentes do receptor, ou com o início de execução do processador receptor, mas assíncrono com o processamento propriamente desejado.

No modelo de mensagens a comunicação tanto pode ser síncrona como assíncrona relativamente ao processador emissor. No primeiro caso, o processador emissor esperará até que o processador receptor receba e execute completamente o pedido feito. No segundo, o processador emissor poderá prosseguir a execução do seu programa logo após o envio da mensagem. Ambas as aproximações têm vantagens e desvantagens. A comunicação síncrona garante a pós-condição do serviço executado no ponto do programa do processador emissor imediatamente após a instrução de comunicação, mas, por outro lado, serializa a execução dos programas associados a esses processadores, reduzindo assim o seu potencial de execução em concorrência. A comunicação assíncrona, por sua vez, potencia a execução concorrente dos dois processadores mas

[11] Nem sempre isso terá de se verificar, como quando, por exemplo se utilizam mecanismos de sincronismo sem bloqueamento.

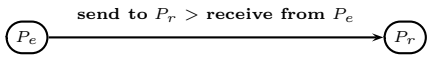

Figura 4.2: Identificação directa.

dificulta a compreensão do efeito conjunto dessa execução. Em contrapartida, uma comunicação assíncrona obriga ao armazenamento temporário das mensagens enviada ao processador receptor numa estrutura do tipo fila, exigência que não se coloca a uma comunicação síncrona, ou a uma comunicação parcialmente assíncrona em que o processador emissor espera até que o receptor comece a executar a mensagem enviada.

No modelo de partilha de memória a comunicação é, por definição, síncrona relativamente à estrutura de dados que representa a memória partilhada (já que a execução é feita pelo mesmo processador), e é assíncrona relativamente a outros processadores que possam utilizar a mesma estrutura partilhada.

4.5.2 Comunicação por mensagens

A comunicação por mensagens entre dois processadores pode ser descrita como a realização de duas operações, uma no processador emissor da mensagem e outra no receptor. Para que a comunicação se realize é necessário que ambos os processadores estejam sincronizados por forma a que a operação de recepção se efectue após a operação de emissão.

Este tipo de comunicação pode ser apresentada da seguinte forma abstracta [Andrews 83, página 25]:

```
SENDER: send EXPRESSION to RECEIVER

RECEIVER: receive VARIABLE from SENDER
```

Assim o processador emissor envia a mensagem `EXPRESSION` para o processador identificado por `SENDER`, e este, por sua vez, recebe a mensagem de `RECEIVER` guardando-a em `VARIABLE`. O conjunto dos identificadores `SENDER` e `RECEIVER` definem um canal de comunicação.

Identificação dos canais de comunicação

O primeiro requisito que esta comunicação exige é uma forma de identificar, no sistema de programação concorrente, os canais de comunicação entre processadores. Existem, para esse efeito, duas alternativas possíveis: ou uma identificação directa ou uma indirecta [Andrews 83].

Na **identificação directa** são associados identificadores a cada processador sendo a comunicação feita expressando directamente os processadores envolvidos (figura 4.2).

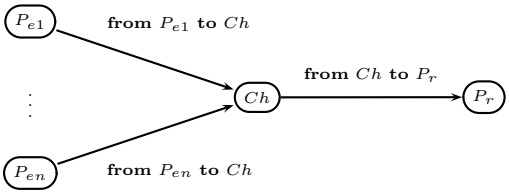

Figura 4.3: Identificação indirecta.

```
program SIMPLE_PRINT_PROGRAM

   process client
     var
       job: PRINTER_JOB
     begin
       loop
         job := fetch_new_job;
         send job to printer
       end
     end

   process printer
     var
       job: PRINTER_JOB
     begin
       loop
         receive job from client;
         print(job)
       end
     end

end -- SIMPLE_PRINT_PROGRAM
```

Esta forma de identificação tem, no entanto, um problema sério que limita grandemente a sua expressividade. No seu uso não é possível expressar a recepção, numa mesma instrução, de mensagens com origem em diferentes processadores emissores.

A outra possibilidade de identificação, denominada de **identificação indirecta**, de processadores consiste na associação de identificadores aos próprios canais de comunicação[12] (figura 4.3). Como é bem visível na figura, com esta forma de identificação dos processadores torna-se possível haver múltiplos emissores para um ou mesmo para múltiplos receptores.

A identificação dos processadores pode também ser classificada como estática ou dinâmica consoante, respectivamente, a identificação dos canais de comunicação entre processadores é apenas possível em tempo de compilação ou se também pode ser feita em tempo de execução.

Na identificação estática de canais de comunicação não é possível expressar canais de comunicação que apenas podem ser conhecidos em tempo de execução, tendo também o problema de os mesmos estarem associados aos processadores durante todo o tempo de vida do programa (mesmo que apenas sejam utilizados num curto espaço de tempo).

Para realizar uma identificação dinâmica de canais pode-se associar tipos de dados aos processadores, na identificação directa, ou aos canais de comunicação na identificação indirecta. Dessa forma torna-se possível criar e destruir dinamicamente os

[12]Designada por vezes como caixas de correio (*Mailboxes*).

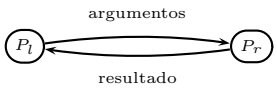

Figura 4.4: Comunicação bidireccional na notação RPC.

canais de comunicação.

Comunicação sequencial de processos

A comunicação sequencial de processos [Hoare 78] (CSP – *Communicating Sequential Processes*) é uma notação de programação concorrente assente na comunicação síncrona e na identificação directa e estática dos canais de comunicação.

A comunicação é feita por comandos de entrada e saída. O comando de saída (emissão) tem a seguinte forma (`destination` é o nome de um processo):

```
destination!expression
```

O comando de entrada (recepção) tem o seguinte aspecto (`source` é também o nome de um processo):

```
source?target-variable
```

O efeito conjunto das duas operações será, caso a operação seja bem sucedida, equivalente à seguinte atribuição de valor:

```
target-variable := expression
```

Temos assim que esta notação de concorrência assenta numa abstracção de atribuição remota de valor (mas restringida, como já foi referido, à identificação directa e estática do canal de comunicação)[13].

Invocação remota de procedimentos

Outra possível notação de comunicação directa entre processadores consiste na abstracção da instrução de invocação de procedimentos (invés da instrução de atribuição de valor como na notação CSP). Esta notação é designada por invocação remota de procedimentos (RPC – *Remote Procedure Call*).

A notação RPC permite uma maior expressividade na comunicação entre processadores já que, ao contrário da notação CSP, permite expressar directamente uma comunicação bidireccional. O processador emissor (local) pode enviar informação para o processador receptor (remoto) através dos argumentos do procedimento e receber informação remota por intermédio do resultado do procedimento (ou seja: no caso de o procedimento ser uma função).

[13] A notação CSP tem outros aspectos importantes, como é o caso da comunicação condicional entre processos, que não iremos apresentar já que não foram importantes na concepção e proposta dos mecanismos concorrentes orientados por objectos.

Nesta notação o nome do procedimento designa o canal de comunicação. Assim, no caso da identificação directa, esse nome será também o nome do processador (cada procedimento remoto estará associado a um processador). No caso da identificação indirecta, terá de existir uma forma alternativa de identificar o processador receptor. Essa identificação pode ser feita, por exemplo, associando um conjunto de procedimentos aos processadores receptores (recorrendo por exemplo ao sistema de tipos da linguagem como acontece com o mecanismo de *rendezvous* da linguagem ADA [Ada95 95]).

4.5.3 Comunicação por partilha de memória

Na comunicação por partilha de memória a comunicação entre processadores faz-se recorrendo a uma estrutura de dados partilhada modificável pelo processador emissor e observável pelo processador receptor. Para que essa comunicação seja bem sucedida é necessário que toda a informação a partilhar seja escrita e lida consistentemente como um todo (ou seja, sem que haja o risco de a informação escrita ser lida de uma forma incompleta ou inconsistente). Para garantir uma consistência temporal na informação partilhada é também desejável que leituras posteriores a uma escrita observem consistentemente o resultado dessa modificação (no próximo capítulo na secção 5.3.1 apresentar-se-á um critério de correcção que garante estas propriedades).

Identificação da memória partilhada

A memória partilhada entre processadores pode ser identificada ou de uma forma explícita – anotando de uma forma distinta as estruturas de dados partilhadas – ou de uma forma implícita – fazendo uso de mecanismos de sincronismo que garantam o comportamento correcto das estruturas de dados por eles sincronizadas.

Muito embora estas duas aproximações pareçam à primeira vista tão só duas formas complementares de observar o mesmo problema, elas de facto representam duas aproximações muito diferentes. Na primeira o sincronismo é implícito (automático) sendo a sua correcção garantida pela semântica das próprias estruturas de dados partilhadas (aproximação axiomática). Na segunda aproximação o sincronismo é explícito (programado directamente pelo programador) sendo a correcção na utilização das estruturas partilhadas garantida pela correcção do programa de sincronismo (aproximação operacional).

A grande vantagem da primeira aproximação é a garantia, em tempo de compilação, da inexistência de erros de sincronização nas estruturas partilhadas, ou seja, a sua segurança (secção 2.2.4). A segunda aproximação, por sua vez, tem a vantagem de poder ser bastante mais flexível e adaptável a novas formas de sincronização[14].

Existem várias linguagens que utilizam uma identificação explícita das estruturas de dados partilhadas como é o caso das linguagens CONCURRENT PASCAL [BH 75] e ADA95 [Ada95 95] (*protected types*). No entanto, sem dúvida que a identificação implícita, apesar dos seus potenciais problemas de segurança, é a mais frequentemente utilizada (por exemplo a biblioteca de POSIX *threads* em C [Butenhof 97])

[14]No próximo capítulo será apresentada uma proposta de sincronização abstracta que, em grande medida, consegue ter ambas as vantagens.

Figura 4.5: Comunicação por partilha de memória e por mensagens.

4.5.4 Relação entre ambos os modelos de comunicação

Em qualquer dos dois modelos de comunicação entre processadores existe uma partilha de informação entre os mesmos. No caso do modelo de partilha de memória a informação é directamente partilhada e utilizável pelos processadores. No caso do modelo de envio de mensagens, a informação é empacotada (eventualmente após ser-lhe retirada uma cópia) sendo enviada conjuntamente com a mensagem. Ou seja, neste caso, a partilha é feita por (eventual) replicação e envio da informação desejada. Existe assim uma dualidade entre os dois modelos de comunicação [Lauer 78] (figura 4.5).

É importante referir-se que muitas vezes pode haver uma mistura dos modelos de comunicação. É o caso de haver partilha de memória na informação enviada entre processadores no modelo por envio de mensagens (INF_1 e/ou INF_2 no exemplo da figura 4.5). Nesta situação, como é evidente, a informação partilhada comporta-se como no modelo de partilha de memória herdando deste todas as suas vantagens e eventuais problemas.

4.6 Sincronização entre processadores

Podemos definir a sincronização entre processadores como sendo o controlo de todas as possíveis interacções entre os respectivos programas por forma não só a evitar a ocorrência de interacções indesejáveis para os programas dos processadores envolvidos, como também a garantir o resultado correcto das interacções desejadas (como é o caso da comunicação entre processadores). Assim um conjunto de processadores estará sincronizado se nos pontos onde pode existir a interferência (desejada ou não) entre as respectivas actividades essa interferência tem resultados controlados, previsíveis e desejados.

A principal aplicação dos mecanismos de sincronismo é, sem dúvida, a comunicação (segura) entre processadores. Nesta situação, os mecanismos de sincronismo devem garantir uma relação causal entre o evento de "execução de uma acção" por parte de um processador e o evento de "detecção dessa acção" por parte dos restantes. Nesta situação o sincronismo pode ser visto como o conjunto de restrições colocadas na ordenação de eventos dos vários processadores [Andrews 83, página 5].

4.6.1 Aspectos de sincronização

Podemos definir três aspectos distintos de sincronização necessários na construção de programas concorrentes:

- interna;
- condicional;

- externa.

A sincronização interna prende-se com a necessidade de uma estrutura de dados partilhada proteger o seu estado interno contra usos inseguros. A sincronização condicional resulta da necessidade de por vezes o acesso a uma estrutura de dados depender do estado da mesma. Por exemplo, o acesso a uma lista para dela retirar um elemento só faz sentido se a lista não estiver vazia. Por fim, a sincronização externa resulta da necessidade de coordenar o uso concorrente de múltiplas estruturas de dados partilhadas por forma a garantir que todas elas são acedidas como se fossem uma única estrutura partilhada.

Esta separação entre estes três aspectos de sincronização é frequente na bibliografia muito embora utilizando denominações alternativas. Holmes [Holmes 97] denomina estes aspectos como sendo, respectivamente, restrições de exclusão, de estado e de transacção[15]. Por vezes, o sincronismo interno é também referido como sincronismo de servidor, e o externo como sincronismo de cliente [Puntigam 05]. Em nossa opinião, a denominação utilizada por Holmes, em particular a de exclusão, não representa devidamente o respectivo aspecto de sincronização, já que existem alternativas para sincronização interna que não obrigam à exclusão mútua dos processadores concorrentes. De qualquer forma, as denominações são substantivamente análogas.

Como seria de esperar, o modelo de comunicação entre processadores é determinante na forma como estes vários aspectos de sincronismo são condicionados.

4.6.2 Sincronização interna

Este aspecto de sincronismo só se coloca, por definição, no modelo de comunicação por partilha de memória. No modelo (puro) de envio de mensagens não há partilha directa de informação, pelo que um bloco de informação só é utilizável (directamente) no máximo por um único processador.

No caso de haver partilha de informação entre vários processadores então torna-se necessário garantir que essa informação não é corrompida por nenhum processador. Para esse efeito existem vários esquemas de sincronismo – desde os mais conservadores que impõem exclusão mútua entre os vários processadores, até aos mais liberais que permitem a utilização em concorrência da informação partilhada – que garantem, sob determinadas condições, a correcção nessa partilha.

Muito embora esses esquemas de sincronismo se apliquem a linguagens procedimentais (e não necessariamente a linguagens orientadas por objectos), optou-se na organização desta tese os apresentar apenas no capítulo 5 (secção 5.10). Dessa forma julgamos tornar mais claras as propostas apresentadas nesta tese.

4.6.3 Sincronização condicional

Como já foi referido, frequentemente o acesso a um recurso partilhado depende não só da necessidade de prevenir erros por competição dessincronizada, mas também da

[15]No seu trabalho de doutoramento [Holmes 99] Holmes identifica outros dois aspectos relacionados com a resposta do sistema concorrente a falhas nas mensagens e com o escalonamento das mensagens. Estes aspectos, não são, no entanto, importantes para o âmbito do nosso trabalho.

verificação de uma determinada condição dependente do estado do recurso partilhado. Por exemplo, um processador de impressão de documentos é obrigado a esperar condicionalmente até que a sua fila de entrada não esteja vazia.

A sincronização condicional terá de estar agregada quer ao sincronismo interno, quer ao externo, aplicando-se aos dois modelos de comunicação entre processadores.

Estratégias de sincronismo condicional

Perante a necessidade de aceder condicionalmente a um recurso partilhado (ou à entrega condicional de uma mensagem), existem basicamente três respostas possíveis caso o recurso não esteja disponível [Lea 00, página 179]:

- reportar a falha imediatamente (*balking*);
- esperar até que a condição se verifique (*guarded suspension*);
- esperar até que a condição se verifique mas apenas num determinado período de tempo (*time-outs*).

Neste trabalho vamo-nos cingir ao caso mais usual de espera condicional até que a condição se verifique.

Modelo de envio de mensagens

No caso de uma comunicação síncrona este aspecto de sincronização obriga[16] o processador emissor a bloquear a sua execução até que a condição de sincronismo seja verificada pelo processador receptor.

No caso de comunicações assíncronas, a espera não se aplica (por definição) ao processador emissor mas sim na fila de espera de mensagens do processador receptor. Um aspecto importante a ter em consideração neste caso, tem a ver com as restrições impostas à ordem das mensagens na fila de espera. Sendo à partida aceitável que mensagens com origem noutros processadores possam passar à frente duma mensagem em espera condicional (até para que a condição de espera possa ser alterada), já o mesmo não se pode dizer relativamente a mensagens com origem no mesmo processador. Caso se permita a alteração na ordem dessas mensagens sem o conhecimento e a anuência do processador emissor, pode-se estar a comprometer o programa do processador emissor caso este dependa da ordem dessas mensagens (o que pode acontecer frequentemente).

Estes aspectos de gestão das filas de espera de mensagens no modelo de comunicação por envio de mensagens, aos quais se acrescentam os que têm a ver com problemas prioridades diferentes (colocados por programas de tempo real), não serão, no entanto, abordados neste trabalho. Consideraremos que a gestão das filas de espera de mensagens é sequencialmente consistente (página 65) o que implica que a ordem das mensagens com origem num determinado processador cliente se mantém no processador servidor.

[16] Pressupondo, como foi referido, a estratégia de espera condicional.

Modelo de partilha de memória

Neste modelo de comunicação a sincronização condicional bloqueia o processador até que seja garantido o acesso exclusivo à estrutura de dados partilhada num estado em que a condição de espera se verifique.

Tanto os esquemas de sincronismo interno, como os de sincronismo externo, são directamente afectados por este aspecto de sincronismo, tendo de haver, por essa razão, uma forte ligação entre eles.

Tal como no caso dos esquemas de sincronismo interno (e também por causa disso), optou-se por apresentar mais detalhadamente o sincronismo condicional no capítulo 5 (secção 5.11).

4.6.4 Sincronização externa

O último aspecto de sincronismo refere-se à necessidade de actuar simultaneamente num conjunto de estruturas de dados partilhadas sem que haja interferências de outros processadores. Para este fim existem basicamente duas aproximações. Uma assenta na reserva para uso exclusivo de todas essas estruturas de dados. Dessa forma consegue-se actuar atomicamente sobre todas essas estruturas de dados. A outra possibilidade consiste no uso de algoritmos de transacções [Lea 00, página 249]. As transacções têm a vantagem de não obrigarem à reserva exclusiva das estruturas de dados envolvidas, mas, no entanto, obrigam a participação voluntária de todas as estruturas envolvidas assim como a que se preveja a possibilidade de a transacção poder falhar obrigando a sua repetição até que seja bem sucedida.

Neste trabalho iremos adoptar apenas a primeira possibilidade de reserva das estruturas de dados partilhadas.

Selecção das estruturas de dados envolvidas

O sincronismo externo, por definição, envolve geralmente várias estruturas de dados partilhadas. Assim os mecanismos para expressar este tipo de sincronismo (quer explícita ou implicitamente) necessitam de identificar quais as estruturas de dados partilhadas que se pretende reservar.

A forma clássica de se atingir esse fim assenta numa instrução (estruturada) de região critica eventualmente condicional [BH 72].

```
region VAR-LIST do
   STATEMENT-LIST
 end
```

Veremos no próximo capítulo (secção 5.15) outras possibilidades para seleccionar as estruturas de dados.

Capítulo 5

Aproximações à Programação Orientada por Objectos Concorrente

Tendo sido apresentadas, com o detalhe julgado necessário, a programação orientada por objectos sequencial e a programação concorrente procedimental, iremos agora estudar com profundidade e alguma sistematização várias possibilidades de integração de mecanismos concorrentes em linguagens orientadas por objectos.

Como é evidente existem inúmeras possibilidades para integração de mecanismos concorrentes em linguagens orientadas por objectos, não fazendo muito sentido apresentá-las a todas e muito menos sem fazer um esforço de comparar as suas qualidades relativas. Assim, torna-se imperativo, por um lado, identificar claramente os critérios de qualidade de linguagens que se pretendem garantir, e por outro, delimitar as características das linguagens orientadas por objectos que servirão de base para esta integração.

Neste trabalho, como foi sendo indicado, e por vezes justificado, ao longo dos capítulos anteriores, optou-se por estudar mecanismos concorrentes em linguagens orientadas por objectos com as seguintes características:

- serem linguagens orientadas por objectos puras (página 20);
- possuírem sistemas de tipos estáticos (página 14);
- considerem objectos como instâncias de TDA (secção 3.9);
- suportarem mecanismos de programação por contrato (secção 3.12);

Os critérios de qualidade de avaliação e construção de linguagens considerados mais importantes foram os seguintes:

- expressividade (secção 2.2.1);
- abstracção (secção 2.2.2);
- segurança (secção 2.2.4);

- sinergia (secção 2.2.5);
- realizabilidade (página 11).

Este capítulo está organizado da seguinte forma. Após a apresentação de algumas definições básicas, a aproximação à concorrência é feita recorrendo primeiramente aos aspectos de programação concorrente apresentados no capítulo anterior. Seguidamente serão abordados alguns dos mecanismos de linguagens orientadas por objectos (todos eles, apresentados no capítulo 3) que podem interferir negativamente com programas concorrentes. Para resolver esses problemas, estuda-se a semântica que estes devem ter num contexto concorrente tentando tirar proveito dessa situação para o aparecimento de comportamentos sinérgicos (seguros) que façam sentido.

5.1 Definições básicas

Para uma melhor compreensão deste capítulo é importante definir-se alguns conceitos.

5.1.1 Objectos concorrentes

Um *objecto concorrente* é um objecto cujos serviços podem ser requeridos por mais do que um processador em períodos de tempo sobrepostos (concorrentemente), ou em que o processador que invoca directamente um dos serviços e o processador que os executa podem ser diferentes. A primeira situação diz respeito ao modelo de partilha de objectos e a segunda ao de envio de mensagens.

Todos os objectos que não forem concorrentes são *objectos sequenciais*. Do ponto de vista da linguagem e do respectivo sistema de compilação, os objectos sequenciais deverão ser absolutamente equivalentes aos objectos sequenciais de linguagens sequenciais (dessa forma não se perdem as vantagens que lhes estão associadas como a sua segurança e eficiência).

5.1.2 Condições concorrentes

Uma expressão booleana (condição) diz-se concorrente se puder depender, no contexto em que é testada, de outro processador para além do responsável pela execução do teste. Uma condição necessária para uma condição ser concorrente é depender, directa ou indirectamente, de consultas a pelo menos um objecto concorrente. No entanto, esta condição não é suficiente já que pode acontecer, no contexto em que a condição é testada por um processador, que eventuais objectos concorrentes envolvidos estejam reservados para uso exclusivo por parte desse processador (logo, o seu estado nunca poderá ser alterado). Outra situação em que condições envolvendo objectos concorrente podem não o ser, ocorre quando o resultado lógico da expressão não depende dos objectos concorrentes envolvidos (independentemente de estarem ou não exclusivamente reservados para esse processador). Por exemplo, a expressão booleana: $i \geq 0$ *and not buffer.empty*, envolvendo a variável inteira i e um objecto concorrente do tipo lista referenciado por *buffer* no caso em que o valor de i é negativo é sempre avaliado para o valor falso, não sendo por isso uma condição concorrente.

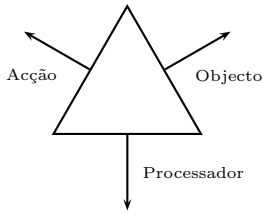

Figura 5.1: As três forças da computação [Meyer 97, página 964].

5.1.3 Asserções concorrentes

Uma asserção diz-se concorrente se a condição que a define for concorrente.

5.1.4 Processadores leitores e escritores

Na execução de um serviço num objecto, vamos designar por **escritor** um processador que está (ou pretende vir a estar) a executar um serviço que pode modificar o estado desse objecto (ou de outros objectos ou entidades externas do programa). Se, pelo contrário, está a executar serviços de consulta puros, então designar-se-á por **leitor**.

5.2 Processadores e objectos

Meyer [Meyer 97, página 964] sustenta que existem três ingredientes básicos da computação: objectos, processadores e acções (figura 5.1). Executar uma qualquer computação será o uso de processadores para aplicar acções em objectos.

No caso de programas concorrentes podemos ter vários processadores a executar acções em objectos.

Em linguagens orientadas por objectos puras todas as acções estão localizadas dentro de objectos (ou pelo menos encapsuladas nas respectivas classes). Nesta situação uma eventual partilha de memória será sempre alcançada dentro de objectos, pelo que no contexto das linguagens concorrentes orientadas por objectos a partilha de memória passará a ser designada por partilha de objectos.

5.2.1 Localização de objectos concorrentes

Para garantir a segurança e a eficiência de programas concorrentes é essencial que o sistema de compilação da linguagem orientada por objectos concorrente seja capaz de identificar todos os objectos concorrentes. Esses objectos requerem que o sistema de compilação lhes associe código de sincronização adequado.

Uma forma segura de identificar esses objectos consiste em usar o próprio sistema de tipos estático da linguagem. Para esse efeito é necessário acrescentar anotações de tipo adequadas que associem inequivocamente as entidades com tipo concorrentes aos

objectos concorrentes. A linguagem SCOOP (apêndice A) consegue esse objectivo utilizando somente a anotação de tipo **separate**. A abordagem seguida para a linguagem protótipo desenvolvida neste trabalho – MP-Eiffel – está descrita na secção 6.5.

5.3 Correcção de objectos

Na secção 3.9 apresentou-se o suporte teórico essencial para a compreensão e correcção de objectos (sequenciais): um objecto é uma instância de uma implementação, possivelmente parcial, de um tipo de dados abstracto (TDA) [Meyer 97, página 142]. Assim, a correcção de um programa depende essencialmente da correcção de cada um dos TDA que implementa, independentemente das possíveis interacções complexas que podem ocorrer entre eles. Temos assim que uma condição necessária para que um objecto esteja correcto é o seu TDA nunca ser comprometido pelo seu uso sequencial ou concorrente.

Em linguagens sequenciais, a imposição de que os objectos só podem ser utilizados nos seus tempos estáveis (página 32) garante a validade do respectivo TDA, sem colocar em causa nenhuma das qualidades importantes dos respectivos programas sequenciais. Essa mesma imposição pode, naturalmente, ser aplicada à programação por objectos concorrente. No entanto, tal implica que no máximo só poderá actuar um único processador dentro de um qualquer objecto. Esta é a situação que ocorre, por definição, em mecanismos de comunicação entre processadores assentes no envio de mensagens, mas que, no caso de mecanismos de comunicação por partilha de objectos, impede a existência de concorrência intra-objecto (ou seja, a possibilidade de vários processadores executarem concorrentemente dentro de um objecto).

Estamos interessados em enfraquecer essa exigência sem, no entanto, se perder a garantia estática de que os TDAs associados aos objectos não são minimamente comprometidos.

Integridade Concorrente de Objectos

A concorrência intra-objecto não pode em caso algum comprometer a implementação do tipo de dados abstracto da respectiva classe.

Uma consequência imediata deste critério é a necessidade de se proibir a existência de atributos públicos modificáveis (página 21). Para garantir minimamente a sanidade semântica dos objectos, esses atributos obrigariam à propagação do sincronismo interno para todos os clientes que pudessem modificar directamente esses atributos.

Este critério assegura que a correcção e integridade de cada objecto individualmente considerado não é comprometida em sistemas concorrentes. No entanto, não é suficiente para garantir a correcção dos próprios sistemas como um todo. Cada processador tem a si associado um programa sequencial que impõe relações de causalidade entre as suas acções. Essa causalidade que não pode, de forma alguma, ser comprometida em programas concorrentes, caso contrário os programas sequenciais associados aos processadores deixam de fazer sentido.

Assim, é necessário garantir também que a ordem das acções imposta pelo programa de cada processador não seja comprometida. Não seria aceitável que de um reordenamento das acções de um processador sobre um objecto resultasse uma inversão da causalidade lógica dessa acções não equivalente à imposta pelo respectivo programa.

Sequencialidade Intra-Processador

A concorrência intra-objecto não pode em caso algum comprometer a causalidade lógica imposta pelos programas de cada processador.

Ou seja, se um processador `P` solicitar a um objecto a realização dos serviços: `s1` e `s2`, nessa ordem, em caso algum o eventual efeito resultante da execução de `s2` no sistema, pode preceder o efeito de `s1`.

Este critério é similar ao chamado critério de consistência sequencial definido por Lamport [Lamport 79].

Consistência Sequencial

Uma execução concorrente de operações sobre um recurso partilhado é sequencialmente consistente se for equivalente a pelo menos um rearranjo sequencial de todas as operações sobre o recurso, em que a ordem de execução das operações em cada processador é mantida.

5.3.1 Linearizabilidade

A consistência sequencial apenas impõe a causalidade de instruções em cada processador, podendo a ordem relativa do processamento de instruções de diferentes processadores variar arbitrariamente. Essa liberdade pode trazer problemas na verificação prática desse critério. Em particular este critério não tem a propriedade de ser local [Herlihy 90b]. Ou seja, a composição de objectos sequencialmente consistentes não garante a consistência sequencial do programa como um todo.

Assim, o critério de correcção considerado apropriado para objectos concorrentes não é a consistência sequencial, mas sim a *linearizabilidade* [Herlihy 87, Herlihy 90b].

Linearizabilidade

Um objecto será linearizável se uma chamada a um qualquer dos seus serviços aparenta ter um efeito instantâneo nesse objecto num qualquer momento entre a invocação e o retorno do serviço.

A linearizabilidade, ao contrário da consistência sequencial, tem a propriedade de ser local. Outra propriedade deste critério com muito interesse é o facto de não obrigar à existência de bloqueamento (como acontece com os monitores e com os esquemas de leitores-escritor). Cria-se assim a possibilidade de utilizar de uma forma segura

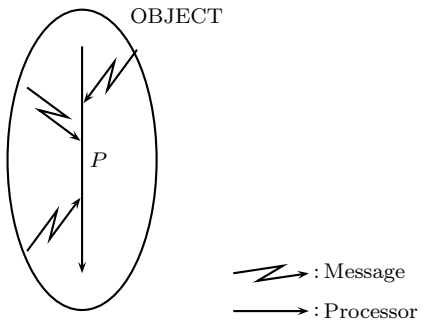

Figura 5.2: Objectos Activos.

mecanismos de sincronismo sem bloqueamento reduzindo ou mesmo eliminando o risco de *deadlocks* e de *starvation*.

Na verificação da linearizabilidade, cada objecto irá ser considerado conjuntamente com eventuais asserções executáveis (invariantes, pré-condições e pós-condições).

5.4 Execução concorrente de processadores

Que possibilidades podem fazer sentido, então, de associação de processadores aos respectivos (sub-)programas em linguagens orientadas por objectos? No capítulo anterior (secção 4.4) foram apresentadas várias possibilidades propostas para linguagens procedimentais.

A aplicação da instrução estruturada de execução concorrente (secção 4.4.1) seria uma possibilidade, mas dadas as limitações de expressividade que ela representa não a iremos considerar.

5.4.1 Associação de processadores a procedimentos

A associação de processadores a procedimentos é uma escolha natural e bem adaptada a linguagens procedimentais. A mesma aproximação em linguagens orientadas por objectos no caso dos procedimentos não pertencerem a nenhum objecto não é aceitável (não sendo mesmo possível, por definição, em linguagens puras). Temos assim que tais procedimentos (ou melhor: rotinas) deverão fazer parte de algum objecto.

5.4.2 Promover os processadores a objectos

Uma possibilidade será fazer com que os processadores sejam também objectos, usualmente designados por objectos activos. Nestes objectos um dos serviços contém o algoritmo do processador e necessariamente também todo o código de sincronização necessário para a comunicação de e para o exterior (figura 5.2). A criação de um desses objectos especiais implica a criação do respectivo processador e a execução integral do seu sub-programa (que está, como foi referido, associado a um único serviço do objecto). Esta é a aproximação seguida pelas linguagens POOL [America 87b], Eiffel// [Caromel 93] e também o Ada [Ada95 95].

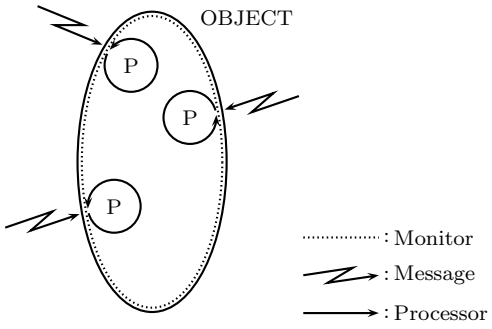

Figura 5.3: Actores.

Esta possibilidade levanta vários problemas. Um deles[1] é o de considerar que esse tipo de processadores é um tipo de dados abstractos válido, o que é de difícil aceitação (seria um tipo de dados abstracto com apenas uma operação). Se esse conceito fosse aplicado ao caso especial de concorrência de um programa sequencial (um processador), torna-se evidente que estaríamos em contradição com a definição básica de programação orientada por objectos (construção de sistemas de software como colecções organizadas de implementações de tipos de dados abstractos).

5.4.3 Associar processadores a objectos

Uma alternativa melhor é apresentada pelo modelo de "actores" [Agha 86, Agha 99] (figura 5.3). Neste modelo, em vez de se considerar os processadores como executando o algoritmo de um único serviço em objectos especiais, faz-se com que aos objectos actores esteja associado um processador (não partilhado com outros actores) capaz de executar um qualquer dos serviços do objecto (temos assim uma associação estática de processadores não a um único procedimento, mas sim a um grupo de procedimentos pertencentes ao objecto actor). Tal como acontecia com a aproximação anterior, um processador é criado conjuntamente com a criação do respectivo objecto actor. Após a sua criação, o processador fica disponível para executar, a pedido de clientes, um qualquer dos serviços públicos do objecto.

Esta aproximação assenta exclusivamente no modelo de comunicação entre processadores por envio de mensagens, e como tal, está bem adaptado à natureza modular distribuída também orientada a mensagens (entre objectos) da programação orientada a objectos. Tem no entanto, a limitação de fazer com que processadores e objectos sejam entidades indissociáveis, impossibilitando a implementação de mecanismos de comunicação por partilha de objectos.

5.4.4 Distribuir objectos por processadores

O passo lógico seguinte será permitir que o mesmo processador lide (em exclusivo) com vários objectos, em vez de apenas um como nos actores (apesar desta generalização, um objecto é sempre executado pelo mesmo processador). É o que acontece na proposta

[1] Outros problemas desta aproximação são sumariamente tratados mais à frente: (página 71) e (secção 5.16).

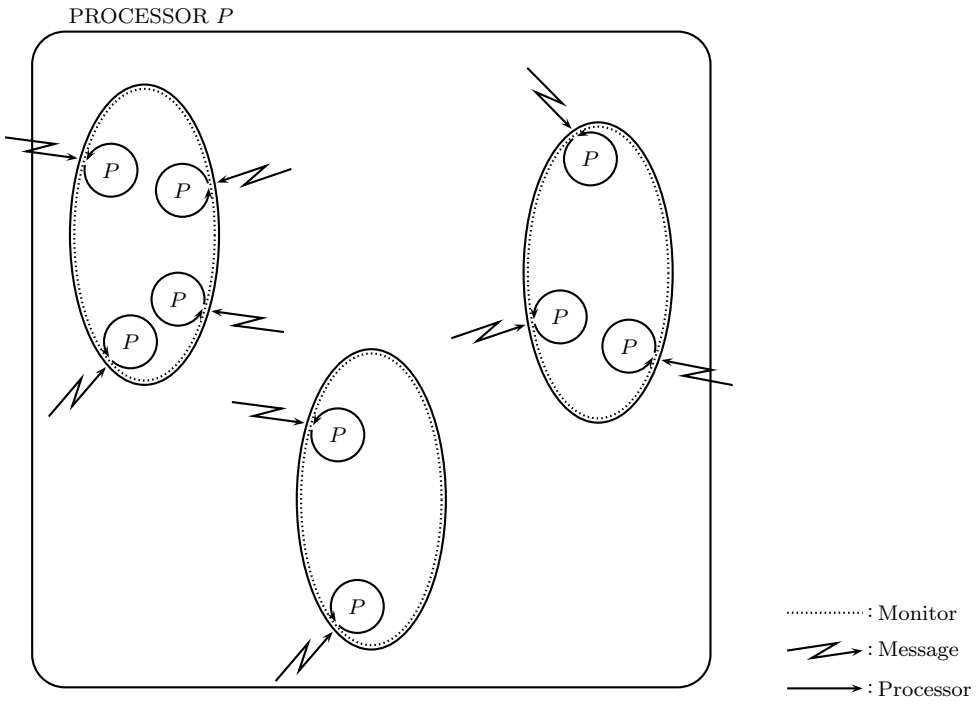

Figura 5.4: SCOOP.

de Meyer (figura 5.4) para incluir concorrência na linguagem EIFFEL [Meyer 97, página 951]: SCOOP[2] (ver apêndice A para uma introdução breve a esta linguagem).

No entanto, tal como acontece na aproximação por actores, esta aproximação tem o problema de restringir a comunicação entre processadores ao modelo de comunicação por envio de mensagens.

5.4.5 Objectos e processadores ortogonais

Uma quarta possível aproximação consiste em fazer com que objectos e processadores sejam entidades completamente independentes. Dito de outra forma, permitir que diferentes processadores possam executar acções nos mesmos objectos, ou seja, ter mecanismos que expressem o modelo de comunicação entre processadores por partilha de objectos. Esta é a aproximação feita em vários sistema concorrentes muito populares, tais como em JAVA e os tipos protegidos do ADA. Se no entanto, esta possibilidade não for feita de forma apropriada, podem-se colocar problemas sérios de segurança.

5.5 Comunicação entre processadores

O aspecto de expressividade mais importante na integração da concorrência em linguagens orientadas por objectos é a relação entre a comunicação entre objectos e a comunicação entre processadores.

[2] *Simple Concurrent Object-Oriented Programming*

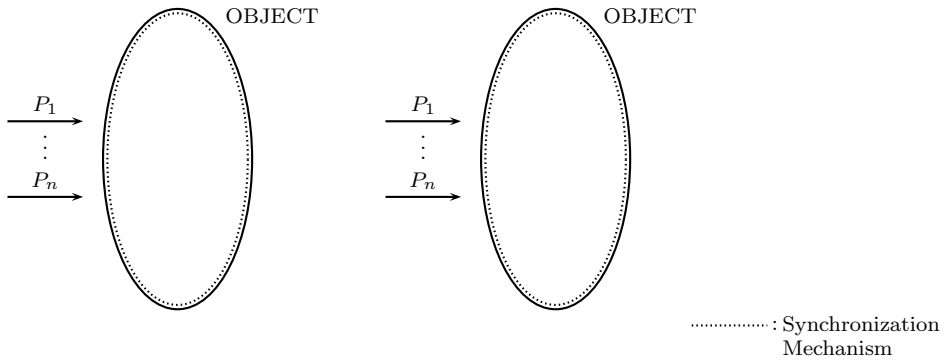

Figura 5.5: Objectos e Processadores Ortogonais.

As linguagens orientadas por objectos utilizam um mecanismo uniforme de comunicação entre objectos assente na passagem de mensagens (página 22). Assim sendo, pareceria natural a sua reutilização como mecanismo de comunicação entre processadores (está de acordo com os critérios usados no projecto de linguagens da abstracção, da segurança, da sinergia, da unicidade e da consistência). No entanto, sendo razoavelmente consensual que a execução de cada processador deve ser semelhante a uma execução sequencial orientada por objectos do respectivo programa (em que o processador vai criando objectos e estabelecendo a comunicação entre eles), o mesmo não terá necessariamente de acontecer com a comunicação entre processadores.

À primeira vista, uma vez que os objectos comunicam entre si através de mensagens, a escolha poderia parecer óbvia: o modelo de comunicação entre processadores por envio de mensagens. No entanto, sendo ambos modelos de comunicação por envio de mensagens, eles aplicam-se a entidades diferentes: objectos e processadores. Sendo assim, o modelo uniforme de comunicação entre objectos por mensagens utilizado em sistemas orientados a objectos é, como se verá, perfeitamente compatível com um qualquer dos dois modelos (ou ambos) de comunicação entre processadores: envio de mensagens e partilha de objectos.

O modelo de comunicação por mensagens entre processadores, numa linguagem orientada a objectos "pura", seria a opção correcta (e única) se cada objecto fosse no máximo executável por um único, e mesmo, processador. É o que acontece com as linguagens do tipo Actores e o SCOOP.

No entanto, esta escolha limita radicalmente as possibilidade de concorrência do sistema, uma vez que impede a existência de concorrência intra-objecto.

Em vez de atribuir cada objecto do programa a um único processador, podemos optar pela sua partilha por mais do que um processador, implementando assim o modelo de comunicação entre processadores por partilha de objectos. A questão não será então pôr em causa o facto de os objectos comunicarem entre si por envio de mensagens (o que acontece sempre), mas sim decidir quais os processadores que têm a responsabilidade de cumprir o pedido executando o serviço apropriado de cada objecto.

Se um programa concorrente for visto como um conjunto de programas sequenciais a comunicar entre si (um por cada processador), então quando há partilha intensiva de recursos a solução mais simples e intuitiva será certamente o modelo de comunicação por partilha de objectos. Por outro lado, em arquitecturas do tipo cliente-servidor, ou

em sistemas distribuídos em que a comunicação entre processadores é baixa, então será mais simples e intuitivo o uso do modelo de comunicação directa entre processadores por envio de mensagens.

Esta é uma das muitas situações em que as regras de projecto de linguagens podem ser consideradas conflituosas, sendo necessário optar pelas mais importantes (sendo que existirá sempre alguma subjectividade e muitos compromissos na escolha feita). Considerando apenas a regra da unicidade (página 12) e também o facto de um qualquer dos dois modelos de comunicação ser implementável com o outro [Lauer 78], a escolha por apenas um deles parece ser a opção correcta. No entanto, como já foi referido, os dois modelos representam duas formas diferentes de expressar a comunicação entre processadores, para necessidades de concorrência em geral diferentes, pelo que, sem as duas hipóteses a linguagem será menos completa, expressiva e simples (indo assim contra o critério de qualidade mais importante da expressividade (secção 2.2.1)).

5.6 Comunicação por envio de mensagens

Debrucemos-nos primeiro sobre possíveis integrações do modelo de comunicação entre processadores por envio de mensagens. Será necessário estudar a forma como este modelo de comunicação pode ser integrado em classes, já que em sistemas orientados a objectos "puros", os processadores só podem existir e realizar trabalho dentro de objectos. Como foi referido no capítulo 4 (página 52), este modelo de comunicação requer a identificação de canais de comunicação entre os processadores. Para se fazer essa identificação existem duas aproximações possíveis: directa ou indirecta. A primeira, na forma como foi descrita na secção 4.5.2, é excessivamente restritiva do lado do receptor pelo que não a iremos considerar. Iremos antes apresentar uma aproximação em que o receptor é directamente identificado pelo emissor sem que, no entanto, a identificação directa inversa se verifique.

5.6.1 Identificação directa do processador destino

Uma possibilidade nesse sentido será associar a cada novo processador um valor enumerável único, por exemplo do tipo inteiro, correspondendo à sua ordem temporal de criação (exemplo em PSEUDO-C na figura 5.6). Esta opção é, no entanto, excessivamente insegura já que não permite garantir, excepto (eventualmente) em tempo de execução, uma comunicação formalmente correcta entre os vários processadores (não seria possível garantir que a informação passada é a esperada pelo processador receptor).

A linguagem ADA – que sem dúvida é uma linguagem interessante – utiliza o sistema de tipos para esta tarefa, incluindo um tipo específico para processadores, no caso designado por **task**.

Com esta estratégia, torna-se possível ter mecanismos de comunicação directa entre processadores de uma forma minimamente segura (embora não completamente segura, já que podem existir problemas de competição dessincronizada no caso de a estrutura de dados passada ao processador ser partilhada).

```
void proc_main(void)
 {
   // proc algorithm
 }

 int main(void)
 {
   int proc;
   Message msg;

   proc = new processor(&proc_main);

   send msg to proc;
 }
```

Figura 5.6: Exemplo de identificação explícita de processadores com um valor inteiro.

No entanto, como aliás é bem visível no exemplo apresentado na figura 5.7, coloca-se o problema sério da decisão sobre quais as mensagens aceitáveis pelo processador receptor. Em ADA essa escolha é feita no programa do processador através da instrução **accept** (eventualmente dentro de um **select** para permitir várias escolhas) aplicável apenas a uma das declarações do tipo **entry** feitas na respectiva especificação.

Não sendo o ADA uma linguagem orientada a objectos "pura" (a versão ADA95 estendeu a versão anterior ADA83 com os mecanismos de herança e polimorfismo, mas na sua essência a ADA95, tal como o C++, é uma linguagem híbrida), poder-se-á alegar que este mecanismo de **tasks** não é orientado a objectos.

De facto, uma situação similar acontece com as linguagens baseadas em objectos activos (secção 5.4.2). Esta opção não é adequada para linguagens orientadas por objectos já que a escolha das mensagens a aceitar pelo processador receptor nada tem a ver com o TDA do respectivo objecto. Pior do que isso, elas são aceites e executadas em tempos não estáveis do objecto pelo que se perde a noção de invariante do objecto e a simplicidade na compreensão e utilização do mesmo. Numa linguagem orientada a objectos, a comunicação com objectos faz-se pela respectiva interface, pelo que não será de estranhar a inadequação da identificação directa do processador destino.

5.6.2 Identificação indirecta

Para realizar uma identificação indirecta, não ambígua, de processadores no contexto de linguagens orientadas por objectos puras, poucas alternativas existirão senão fazer uso dos próprios objectos.

Uma aproximação simples consiste em associar cada objecto, durante todo o seu tempo de vida, a um único processador (que em princípio deverá ser o processador que o criou). Na família de linguagens do tipo "actores" (secção 5.4.3) e no SCOOP (secção 5.4.4) é esta a forma escolhida para identificar processadores. Uma mensagem enviada a um objecto que pertença a outro processador será uma comunicação directa entre os respectivos processadores. Esta opção tem, relativamente à anterior, a vantagem de ser bem adaptada aos sistemas orientados a objectos, evitando as situações

```
-- a_processor.ads
package A_Processor is
  task type Processor is
     entry Start(A_Argument: in Positive);
     entry Another_Rendezvous;
     entry Finish;
  end Processor;
end A_Processor;


-- a_processor.adb
with Ada.Text_IO;
use Ada.Text_IO;
package body A_Processor is
  task body Processor is
     Done : Boolean;
  begin
     accept Start (A_Argument: in Positive) do
       Put_Line("Processor started with argument: " & Positive'Image(A_Argument));
     end Start;
     Done := false;
     while not Done loop
       select
         accept Another_Rendezvous do
           Put_Line("Rendezvous...");
         end Another_Rendezvous;
       or
         accept Finish do
           done := true;
         end Finish;
       end select;
     end loop;
  end Processor;
end A_Processor;


-- main.adb
with Ada.Text_IO;
use Ada.Text_IO;
with A_Processor;
procedure Main is
  proc: A_Processor.Processor;
begin
  proc.Start(10);
  proc.Another_Rendezvous;
  proc.Finish;
end Main;
```

Figura 5.7: Exemplo de identificação explícita de processadores com o sistema de tipos.

muito problemáticas de poderem existir comunicações entre processadores em alturas em que o invariante do objecto do processador receptor (ou seja, aquele que terá de processar a mensagem) pode não se verificar. Neste caso os processadores receptores só responderão quando o respectivo objecto estiver num tempo estável, o que reduz drasticamente a complexidade dessas interacções.

Esta aproximação é similar a uma invocação remota de procedimentos (página 54) aplicada a serviços públicos dos objectos, com a vantagem da escolha dos serviços a ser remotamente invocados ser devidamente contextualizada pelo TDA dos objectos (ou seja, tirando proveito da metodologia orientada por objectos).

5.6.3 Comunicação síncrona e assíncrona

No capítulo 4 (secção 4.5.1) referiu-se que, neste modelo, a comunicação tanto poderia ser síncrona como assíncrona. Do ponto de vista da expressividade da linguagem ambas podem ser úteis. A comunicação assíncrona aumenta a concorrência do programa já que permite que o processador emissor continue a execução do respectivo algoritmo independentemente do processador receptor. Por outro lado, a comunicação síncrona garante a pós-condição do serviço executado remotamente logo após o envio da mensagem, o que pode ter consequências importantes na garantia de correcção do algoritmo.

Uma sinergia muito interessante pode ser retirada se se tiver em consideração a diferença semântica entre serviços do tipo comando e do tipo consulta (página 19). Com efeito, a invocação de um comando pode ser considerada uma comunicação dirigida unicamente do cliente para o objecto, pelo que se adapta perfeitamente a uma comunicação assíncrona (excepto no que diz respeito à verificação da pré-condição como veremos à frente). Já a invocação de uma consulta sobre um objecto é uma comunicação bidireccional pelo que se justifica que deverá ser síncrona.

Caromel [Caromel 89, Caromel 93] propõe uma alternativa, designada de "espera por necessidade"[3] em que a espera não é feita imediatamente na invocação dos serviços de consulta, mas sim apenas quando o respectivo resultado é necessário. Meyer, na extensão SCOOP [Meyer 97, página 987], adoptou a mesma ideia. No entanto, este mecanismo de espera por necessidade pode interferir negativamente com outros mecanismos das linguagens, em particular com os mecanismos de suporte à programação por contrato. A interferência potencialmente mais gravosa ocorre com a verificação da pré-condição do serviço remotamente invocado (no caso, obviamente, de essa pré-condição existir). Com efeito uma falha na pré-condição é da responsabilidade do cliente (e não do objecto), pelo que permitir que a verificação desta asserção seja assíncrona com o programa do processador cliente tem efeitos extremamente negativos. Desde logo, perde-se a possibilidade de sinalizar, através de uma excepção, no ponto apropriado do programa desse processador a falha que é da responsabilidade desse mesmo processador. O resultado desta situação é a degradação da robustez do programa, podendo mesmo inviabilizar a implementação de algoritmos adequados de tolerância a falhas. Por estas razões parece-nos que, independentemente do tipo de comunicação assíncrona (seja por invocação de um comando, ou devido ao mecanismo de espera por necessidade), é

[3] *wait-by-necessity*.

obrigatório impor a verificação síncrona da pré-condição[4].

No caso da aplicação da espera por necessidade a serviços de consulta levanta-se também o problema da verificação da pós-condição do serviço e do invariante do objecto. Esta situação é bastante menos gravosa que no caso das pré-condições, já que se pode aceitar que a eventual excepção (a ser propagada para o cliente) possa ser entregue no ponto de espera (em vez de ser no ponto de invocação). Será uma semântica aceitável para a situação, embora possa causar problemas já que os programas dos processadores clientes terão, eventualmente, de replicar o código de gestão de falhas para vários localizações (todas as que podem esperar resultados da invocação inicial).

A justificação mais importante para a adopção deste mecanismo de espera por necessidade assenta no aumento do potencial de concorrência do programa, já que os processadores clientes podem continuar a sua agenda sem esperar "desnecessariamente"[5] pelo outro processador. No entanto esse problema só se coloca no caso da linguagem adoptar apenas o modelo de comunicação entre processadores por envio de mensagens. No caso da linguagem adoptar os dois modelos (como a nossa proposta apresentada no próximo capítulo), então o potencial de concorrência do programa pode ser maximizado pelo modelo de partilha de objectos. Neste último modelo, a comunicação é síncrona, pelo que não provoca nenhuma destas interferências negativas com o mecanismo de excepções (secção 5.18).

5.7 Comunicação por partilha de objectos

A aplicação deste modelo de comunicação na programação concorrente procedimental (secção 4.5.3) faz-se recorrendo a estruturas de dados partilhadas. Numa integração orientada por objectos obviamente que essas estruturas de dados terão de ser substituídas por objectos partilhados. No entanto, é muito importante ter em conta que os objectos não são estruturas de dados (secção 3.4). Sendo a programação por objectos imperativa, é usual os objectos terem a si associadas estruturas de dados. No entanto, elas são internas ao objecto e os serviços do objecto podem não se aplicar exclusivamente a essa estrutura interna (podendo ter efeitos colaterais, nem sempre reversíveis, em outros objectos ou mesmo em entidades exteriores ao próprio programa). Estas características típicas dos objectos (mas inexistentes nas estruturas de dados) podem afectar a realizabilidade de implementações seguras de objectos partilhados (estes problemas serão tratados nas secções de sincronismo intra-objecto 5.10).

Um aspecto interessante na integração deste modelo de comunicação é que ele partilha uma característica muito importante com a comunicação entre objectos das linguagens sequenciais: o processador que requer a execução de um serviço de um objecto é o mesmo que depois vai executar esse serviço. Ou seja, muito embora seja habitual o uso da terminologia de envio de mensagens entre objectos nas linguagens orientadas por objectos, de facto o modelo de comunicação entre processadores por partilha de objectos é, neste aspecto, mais natural do que o modelo de comunicação entre processadores por envio de mensagens.

[4] Apenas no que diz respeito à parte sequencial da pré-condição, já que a parte concorrente (a existir) tem outra semântica como se verá mais à frente (secção 5.14).

[5] As aspas justificam-se porque de facto a espera pode mesmo ser necessária.

Veremos que para muitos dos mecanismos das linguagens orientadas por objectos, como por exemplo o mecanismo de excepções (secção 5.18), este modelo de comunicação permite que o seu comportamento seja semelhante ao das linguagens sequenciais.

No entanto, a comunicação por partilha de objectos, quando comparada com o modelo por envio de mensagens, em geral dificulta a sincronização dos objectos partilhados. Este problema será abordado na secção 5.10.

5.8 Integração de ambos os modelos de comunicação

Pode-se optar por adoptar apenas um dos modelos de comunicação – envio de mensagens (Actores, SCOOP) ou partilha de objectos (Java) – ou então optar por ambos (Ada95). As linguagens de programação servem como meios para resolver problemas computacionais. Assim, na ponderação sobre qual a melhor escolha, a primeira pergunta a que devemos dar resposta será qual das três possibilidades facilita o trabalho dos programadores. É claro que a resposta a essa pergunta pode depender do domínio de aplicação requerido por cada programador.

Para se expressarem algoritmos em linguagens de aplicação geral, não haverá dúvidas que ambos os modelos são úteis consoante os programas a desenvolver. No caso do modelo por envio de mensagens, ele adequa-se bem a programas concorrentes em que os processadores estão pouco ligados entre si (por exemplo, em sistemas distribuídos cliente-servidor). Já o modelo por partilha de objectos é bem adaptado a programas concorrentes em que os processadores estão fortemente ligados, com partilhas frequentes de objectos.

É claro que, como já foi referido, é sempre possível converter programas expressos num modelo para o outro. No entanto, essa conversão será, em geral, feita à custa de uma menor expressividade e uma menor eficiência. Assim, é nossa opinião que é desejável ter mecanismos para ambos os modelos em linguagens orientadas por objectos concorrentes de aplicação geral.

5.8.1 Interfaces distintas?

Nessa situação, coloca-se o problema de ser aceitável que se use a mesma interface (ou seja, a mesma perspectiva do TDA do objecto) para ambas as formas de comunicação.

À partida parece-nos que, nessa situação, as interfaces não devem necessariamente ser as mesmas. As duas formas de comunicação envolvem um comprometimento muito diferente dos processadores eventualmente envolvidos. Na comunicação por envio de mensagens é obrigatória a colaboração directa de pelo menos dois processadores, pelo que nos parece excessivo obrigar o processador receptor a ter que responder à invocação de um qualquer dos seus serviços públicos na sua interface normal. Aliás essa situação não se aplica somente a este caso de comunicação concorrente. No caso da criação de objectos, a larga maioria dos serviços públicos do objecto não pode ser utilizada como eventual serviço de inicialização do objecto.

Parece-nos que a integração mais adequada é a reutilização da interface normal dos objectos para a comunicação por partilha de objectos, e possibilitar a definição de uma

interface separada (partilhando os serviços do objecto) para a comunicação por envio de mensagens.

5.9 Sincronização entre processadores

A sincronização entre processadores – essencial, entre outras coisas, para que estes possam comunicar entre si – é, sem dúvida, o requisito que tradicionalmente mais problemas tem levantado à integração de concorrência em linguagens orientadas por objectos [Holmes 98, Briot 98]. É opinião do autor que uma grande parte destes problemas se deve ao uso de mecanismos com sincronismo explícito (página 55), ou seja, recorrendo a uma aproximação operacional à sincronização colocando nas mãos do programador a responsabilidade de sincronizar correctamente os objectos concorrentes.

Pretendemos seguir a abordagem alternativa do sincronismo implícito (ou automático) numa aproximação explícita à concorrência (secção 4.1.1). É claro que esta aproximação ao problema obriga, não só a ter-se mecanismos na linguagem que expressem e abstraiam adequadamente a comunicação entre processadores[6] (causa primeira para a necessidade de sincronismo), como também a que se verifique a realizabilidade de possíveis implementações automáticas (ou seja, a realizar pelo sistema de compilação) de esquemas de sincronismo apropriados e correctos.

5.9.1 Sincronização abstracta

Uma abordagem automática ao sincronismo de objectos concorrentes pode ter a desvantagem de pouca adaptabilidade do esquema de sincronismo a diferentes situações e necessidades. De facto, se se restringir estaticamente a sincronização de um objecto concorrente partilhado, por exemplo, à exclusão mútua na execução dos seus serviços, podemos estar a excluir usos concorrentes perfeitamente seguros do objecto, como por exemplo permitindo vários processadores leitores de observarem o seu estado.

Por outro lado, se a sincronização de objectos for da responsabilidade do programador, corre-se o risco, potencialmente bastante mais grave, de se construírem objectos incorrectamente sincronizados.

Numa aproximação segura (secção 2.2.4) a uma linguagem concorrente é essencial que a correcção nos mecanismos de sincronismo não dependa, de forma alguma, do programador. Numa aproximação segura e abstracta (secção 2.2.2) a uma linguagem concorrente, para além da exigência anterior, será essencial que o programador possa escolher um qualquer esquema de sincronismo desde que ele seja garantidamente seguro e realizável pelo sistema de compilação. As várias abordagens possíveis à escolha dos esquemas de sincronismo serão tratadas na secção 5.10.10.

5.9.2 Aspectos de sincronização

No contexto da programação concorrente orientada por objectos, os aspectos de sincronização definidos na secção 4.6.1 são melhor descritos com os seguintes termos:

[6] Uma proposta nesse sentido será apresentada no capítulo 6 no âmbito da linguagem protótipo avançada neste trabalho.

- intra-objecto (interna);
- condicional;
- inter-objecto (externa).

Nas secções seguintes vamos estudar a realizabilidade automática destes vários aspectos de sincronização incluindo a integração automática de todos estes aspectos no mesmo objecto concorrente.

5.10 Sincronização intra-objecto

Este aspecto de sincronização, como já foi referido (secção 4.6.2), aplica-se ao modelo de comunicação por partilha de objectos.

Nesta secção vai-se apresentar vários esquemas de sincronismo identificando, para cada um deles, as condições de realizabilidade colocadas à sua implementação automática por sistemas de compilação de linguagens concorrentes.

5.10.1 Disponibilidade concorrente de objectos

Por forma a comparar-se diferentes esquemas de sincronização intra-objecto, é útil ter algum tipo de métrica objectiva que indique o potencial máximo de concorrência de um objecto. Esse é o propósito da métrica *disponibilidade concorrente de objectos*.

Considerando que N_x é o número máximo de processadores que partilham uma qualquer propriedade x (por exemplo: leitor ou escritor) pretendendo operar num objecto, e que N_c é o número máximo destes processadores que lá podem actuar concorrentemente em segurança ($N_c <= N_x$), define-se a disponibilidade concorrente desse objecto (COA_x[7]) relativamente aos processadores com a propriedade x como sendo:

$$COA_x[\%] = \frac{N_c}{N_x} \tag{5.1}$$

Este factor mede a percentagem máxima de processadores com uma determinada propriedade que podem operar concorrentemente com segurança dentro de um objecto.

Chama-se a atenção que este valor não é necessariamente único em cada esquema de sincronismo, podendo depender do estado concorrente do objecto (por exemplo, o uso de um objecto por processadores com uma determinada propriedade pode excluir o seu uso por processadores com outras propriedades).

5.10.2 Cobertura total de objectos

Uma imposição necessária para que um qualquer mecanismo de sincronismo intra-objecto possa ser aplicado com segurança a objectos, é a necessidade de todos os serviços exportados do objecto estarem sincronizados[8].

[7] *Concurrent Object Availability*

[8] Em JAVA [Lea 00, página 78] os objectos com esta propriedade designam-se por objectos completamente sincronizados ou atómicos.

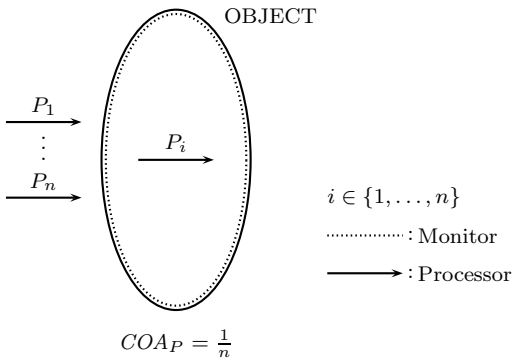

Figura 5.8: Monitores.

Cobertura total de objectos

É condição necessária de correcção no sincronismo de objectos partilhados que todos os seus serviços não estritamente privados estejam sincronizados com algum mecanismo.

Uma das objecções fortes [BH 99] aos mecanismos de concorrência, de base, da linguagem JAVA reside precisamente em não existir uma garantia de cobertura total no sincronismo de objectos, uma vez que caso não se verifique esta condição poderão existir problemas por competição dessincronizada.

5.10.3 Monitores

Uma aproximação simples e suficiente para garantir a linearizabilidade consiste em considerar cada objecto como sendo um monitor [Hoare 74] (figura 5.8). É aliás interessante constatar que os próprios Hoare [Hoare 74] e Brinch Hansen [BH 93] reconheceram a importância do conceito de classe da primeira linguagem orientada por objectos – SIMULA – quando propuseram os monitores.

Os monitores são o mais simples de todos os esquemas de sincronismo intra-objecto. O preço a pagar por essa simplicidade é o facto de os monitores só estarem disponíveis para um processador de cada vez. Para n processadores o valor COA de um monitor é de $\frac{1}{n}$, o que é o menor valor útil possível.

O mecanismos de concorrência da linguagem JAVA foram inicialmente pensados para serem aproximações de monitores [Gosling 96, página 399], mas os seus intentos falharam em alguns aspectos importantes [BH 99]. A versão actual da linguagem [Gosling 05], embora não resolva alguns dos problemas de base com os monitores, permite a utilização de outros esquemas de sincronismo para além de monitores[9] [Lea 00].

Realizabilidade

Os monitores colocam relativamente poucas condições sobre os sistemas de compilação. Uma exigência elementar de base[10] é a necessidade de serem identificados

[9] Mantendo, no entanto, uma aproximação explícita ao sincronismo.

[10] Para além, é claro, da identificação dos objectos concorrentes (secção 5.2.1).

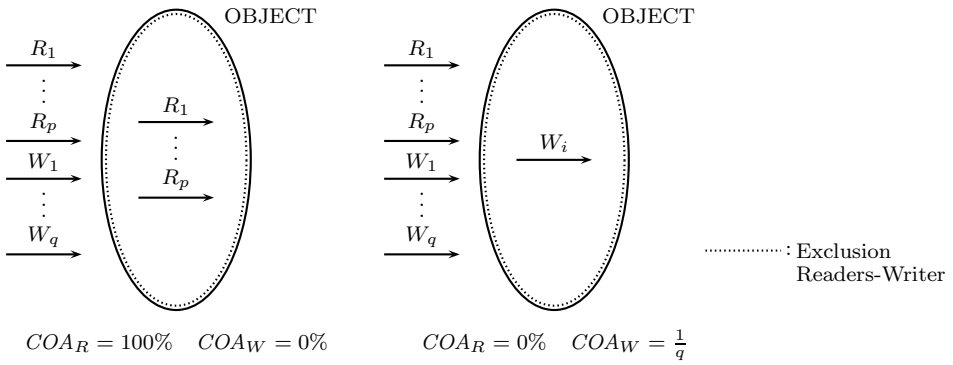

Figura 5.9: Exclusão entre Leitores-Escritor.

todos os serviços públicos do objecto. Esses serviços necessitam de ser protegidos com o código de sincronismo do monitor.

Um algoritmo possível para implementar este esquema de sincronismo consiste em criar uma nova classe que encapsule a classe não sincronizada, mantendo a mesma interface, e na qual o código de sincronismo do monitor é implementado. Esta possibilidade tem a vantagem de evitar o problema de sobre-sincronização (sincronização repetida ou recursiva) na chamada de serviços públicos dentro do próprio objecto. A secção C.1.2 apresenta, como exemplo, uma possível implementação automática do esquema de sincronismo por monitor de uma estrutura do tipo pilha (`LIFO`[11]) que, por sua vez, é apresentada na secção C.1.1. Como é fácil de verificar, a sincronização automática da classe do tipo pilha (para a classe MONITOR_STACK) exige pouco conhecimento semântico sobre a classe não sincronizada por parte do sistema de compilação. Muito embora o algoritmo de sincronização condicional (o qual será descrito na secção 5.11) lá existente tire proveito da capacidade de distinguir comandos e consultas impuras de consultas puras, tal não é uma exigência dos monitores mas tão só uma optimização deste algoritmo.

5.10.4 Exclusão entre leitores-escritor

A imposição de exclusão mútua no processamento de serviços de objectos pode ser considerada uma restrição excessiva. Frequentemente, alguns dos processadores estão apenas a tentar consultar (sem efeitos colaterais) o objecto para obter determinada informação. Nestes casos, é suficiente garantir a exclusão mútua quando está a ser processado algum serviço que possa modificar o estado do sistema (ou do próprio objecto ou de outros), permitindo o processamento concorrente dos restantes serviços (consultas puras).

Portanto uma aproximação utilizando o esquema de sincronismo leitores-escritor [Courtois 71] (um escritor exclui todos os outros processadores, mas múltiplos leitores podem concorrentemente aceder ao objecto) é também uma opção válida e segura (figura 5.9). Este esquema tem um valor *COA* médio maior do que o dos monitores, sendo assim menos sujeito a bloquear o acesso a objectos concorrentes, o que pode reduzir o risco de existirem alguns problemas de *liveness* como os *deadlocks*.

[11] *Last In First Out.*

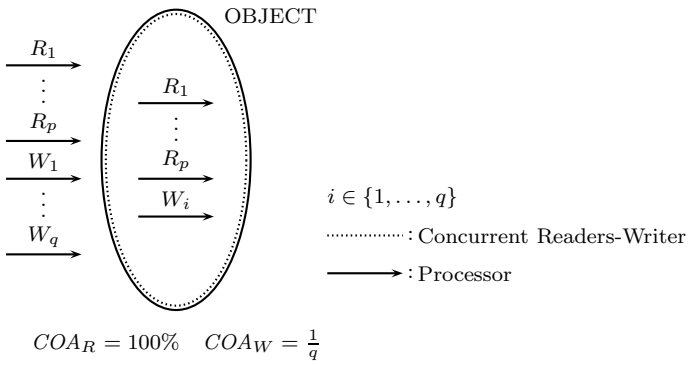

Figura 5.10: Leitores-Escritor Concorrentes.

Este esquema de sincronismo é utilizado na linguagem Ada95 (tipos protegidos), e foi também a aproximação inicial tomada na linguagem MP-Eiffel proposta pelo autor [OeS 04] (modificada mais tarde para uma sincronização abstracta [OeS 06a]).

Realizabilidade

Com utilização deste esquema ganha-se na disponibilidade concorrente dos objectos mas o sistema de compilação necessita de extrair mais informação das classes a sincronizar. Ao contrário dos monitores, este esquema requer a capacidade de distinguir comandos e consultas impuras de consultas puras. No apêndice B, secção B.3 descreve-se informalmente o algoritmo seguido na linguagem MP-Eiffel para resolver esse problema.

Muito embora o sincronismo de exclusão leitores-escritor tenha uma menor contenção do que os monitores, tem, no entanto, uma implementação mais pesada do que um mecanismo simples de exclusão mútua, penalizando (ainda que muito ligeiramente) a eficiência sequencial de cada processador no acesso a serviços que modificam o objecto. Este aspecto, que se verificará também noutras escolhas de esquemas de sincronismo, é similar aos problemas de optimização existentes em linguagens sequenciais. Assim, o ideal será provavelmente o sistema concorrente não impor obrigatoriamente uma implementação em particular, mas sim garantir um comportamento correcto, deixando o trabalho de escolha sobre a forma como são implementados a um sistema de optimização do sistema de compilação. Ou seja, também aqui a opção pela sincronização abstracta mostra ser correcta.

A secção C.1.3 apresenta uma possível realização automática deste esquema de sincronismo para uma pilha.

5.10.5 Leitores-escritor concorrentes

Lamport [Lamport 77] propôs uma generalização ao esquema de sincronismo anterior, que permite o acesso concorrente entre múltiplos serviços de "leitura" e um serviço de "escrita". A exclusão mútua é apenas necessária relativamente a múltiplos processadores escritores (figura 5.10). Desta forma, os processadores leitores nunca bloqueiam um possível processador escritor. Nesta proposta de Lamport, os serviços de consulta

terão de ser repetidos sempre que ocorrem em concorrência com um processador escritor.

Na integração deste esquema de sincronismo em objectos, é necessário prever a situação em que o invariante dos objectos não se verifica no início, ou no fim, da execução de serviços de consulta devido, simplesmente, a uma execução concorrente de um escritor. Essa situação tem de ser devidamente tratada, fazendo com que quebras do invariante, ou de qualquer outra asserção ocorridas antes ou após a execução de serviços de leitura, e caso tenha havido ou esteja a haver uma escrita concorrente, resultem na repetição (transparente, no comportamento do programa) da execução desses serviços. Se, pelo contrário, a falha numa dessas asserções ocorrer sem que haja uma execução concorrente de um escritor, então uma excepção tem de ser gerada como será de esperar na utilização de um objecto incorrecto.

Este esquema de sincronismo é muito interessante pelo facto de impor, em termos de implementação, poucas mais restrições do que o esquema de exclusão leitores-escritor. Tem uma menor contenção (um *COA* relativamente maior ou, no pior caso, igual) na execução dos processadores escritores, o que reduz o risco de *deadlocks*. No entanto, pode criar problemas de *starvation* nos processadores leitores quando a execução dos serviços de escrita é excessivamente frequente [Lamport 77, Peterson 83].

Uma solução possível, em certos casos, para este problema é proposta por Peterson [Peterson 83]. A ideia base assenta na duplicação dos dados partilhados (que, neste caso, seria a duplicação do estado dos objectos). No caso particular importante em que existe apenas um processador escritor, Peterson [Peterson 83] propõe um algoritmo sem espera para qualquer processador (ou seja, $COA = 100\%$).

Realizabilidade

Este tipo de sincronismo mantém as restrições impostas ao esquema anterior, estendendo-as com a necessidade de as operações de leitura poderem ter de ser repetidas no caso de falha (ou seja, sempre que há uma escrita concorrente).

Esta repetição (escondida dos clientes dos objectos), não levanta problemas sérios de implementação, nem no comportamento esperado dos objectos porque, por definição, os serviços de consulta puros não alteram o estado dos objectos. No entanto, como foi referido, é necessário prever a situação em que ocorrem falhas de asserções na execução por um processador leitor como resultado de alterações no estado do objecto devidas a um processador escritor. Assim, este esquema de sincronismo requer uma linguagem na qual seja possível apanhar, transparentemente, todas as excepções geradas durante a execução de serviços de consulta, permitindo que se verifique se a causa da falha se deve a uma interferência com um processador escritor concorrente – caso em que a excepção pode ser ignorada e a execução do serviço repetida – ou se é de facto uma falha real numa asserção. Esta restrição é essencial para que se possa implementar correctamente este mecanismo, já que só assim há a possibilidade de distinguir as falhas reais das resultantes de competições dessincronizadas (neste caso em particular, inócuas).

Este problema de quebra temporária do invariante pode ser completamente evitado no caso particular de existir apenas um processador escritor. Nesta situação existem algoritmos, como o de Peterson [Peterson 83], em que os processadores leitores observam sempre o objecto partilhado num estado estável.

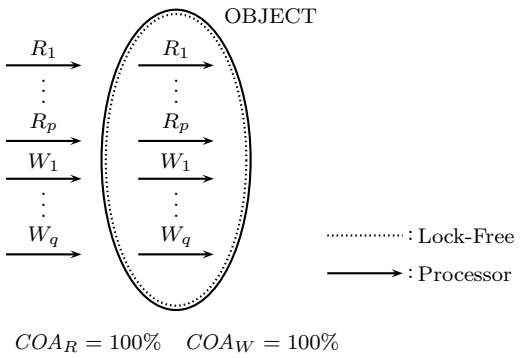

Figura 5.11: Sincronismo Sem Bloqueamento.

A secção C.1.4 apresenta uma possível realização automática deste esquema de sincronismo para uma pilha.

5.10.6 Sincronismo sem bloqueamento

Um grupo de esquemas de sincronização que vem merecendo um interesse crescente é o chamado sincronismo sem bloqueamento [Herlihy 91] (figura 5.11). Este tipo de sincronismo é caracterizado por garantir que os processadores conseguem executar operações numa estrutura de dados partilhada independentemente dos tempos de execução de outros processadores, e que pelo menos um deles será sempre bem sucedido. Um caso particular importante é o sincronismo sem espera em que é garantido que todos os processadores conseguem realizar a operação desejada em tempo finito.

As vantagens deste esquema assentam na inexistência de bloqueamento de processadores[12] (pelo que são imunes a *deadlocks*) e na sua tolerância a falhas de outros processadores. Estas características fazem com seja especialmente apropriado para sistemas de tempo real [Anderson 97].

Actualmente este tipo de sincronismo é pouco utilizado, embora seja previsível alguma mudança nessa situação. Um sinal disso foi o lançamento público de uma biblioteca de classes para Java que faz uso deste sincronismo (*JSR 166: Concurrency Utilities* [Sun Microsystems Java Specification Requests 04]).

As razões que levam a que este tipo de sincronismo seja tão pouco utilizado são a sua complexidade, a especificidade de muitos dos seus algoritmos, e principalmente a dificuldade em garantir implementações seguras.

Nesta secção estamos interessados apenas numa aproximação preliminar para futuras implementações automáticas seguras destes esquemas. É importante referir que, à parte alguma experimentação com algoritmos sem bloqueamento em C, não foi feita nenhuma experimentação com estes esquemas na linguagem protótipo proposta.

Noções Básicas

Em geral, os algoritmos de sincronismo sem bloqueamento baseiam-se na duplicação total ou parcial das estruturas de dados partilhadas (objectos, neste caso) e, quando necessário, na concentração num único instante atómico de todas as modificações a

[12] Apenas para o aspecto de sincronismo intra-objecto.

essa estrutura de dados requeridas por cada operação. Essa modificação atómica do estado do objecto recorre, geralmente, a instruções especiais de *hardware*, tais como as instruções `CAS` – *Compare-And-Swap* – ou `LL/SC` – *Load-Linked, Store-Conditional*). Nesses algoritmos, tal como acontecia com o esquema de sincronismo de leitores-escritor concorrentes, é necessário prever a possibilidade de falhas na actualização do estado do objecto, devidas à acção de outros processadores concorrentes. Nesses casos, é necessário repetir todo o processo (até que seja bem sucedida). No caso especial dos algoritmos sem espera, como já foi mencionado, é garantido um limite máximo ao número de repetições.

Herlihy [Herlihy 90a, Herlihy 91] demonstrou que existem algoritmos universais capazes de implementar este sincronismo em objectos concorrentes respeitando o critério da linearizabilidade, tendo também apresentado metodologias universais (embora não muito eficientes) [Herlihy 90a, Herlihy 93] para a sua implementação. A metodologia apresentada, como é referido por Herlihy, é passível de ser realizada automaticamente pelo sistema de compilação.

Outros possíveis esquemas relacionados com o sincronismo sem bloqueamento são baseados em sistemas de transacções de memória por *software*[13] [Herlihy 03]. Estes algoritmos funcionam de forma similar às transacções em sistemas de bases de dados. As transacções processam-se em três passos. Primeiro a transacções é enunciada, depois é feita a execução das operações requeridas e, finalmente, é feita uma tentativa para submeter o resultado da transacção. Caso essa submissão falhe, é garantido que a tentativa de transacção não modificou o estado do objecto, podendo ser novamente tentada. Caso seja bem sucedida, o resultado da transacção tomará efeito (atomicamente) no estado do objecto. Este processo de transacção é repetido até que seja bem sucedido. Harris e Fraser [Harris 03] propõem um mecanismo para a linguagem Java (fortemente baseado nas regiões criticas condicionais de Hoare) que tira vantagem das possibilidades oferecidas pelos sistemas de transacções de memória por *software* para algoritmos sem bloqueamento (a proposta também inclui um mecanismo para sincronização condicional). Se as exigências impostas sobre o sistema de compilação, apresentadas à frente, forem observadas, a proposta de Harris e Fraser pode, em princípio, ser usada para implementar este esquema de sincronismo. Para que tal seja possível, no entanto, é requerido que o sincronismo seja aplicado a todos os serviços públicos do objecto. Como já foi referido atrás, a possível adopção futura destes esquemas de sincronismo exigirá previamente uma adequada experimentação.

Realizabilidade

Quer o algoritmo genérico de Herlihy [Herlihy 93], quer os algoritmos de transacções de memória por *software*, requerem a capacidade de se retirar cópias do estado dos objectos, e a possibilidade de haver possíveis repetições na execução de serviços. É este último requisito que mais restrições impõe a realizabilidade estática destes algoritmos.

De facto, mesmo tendo em conta que a execução de um serviço por um processador é aplicada a uma cópia estável separada do objecto, nem todos os serviços podem ser repetidamente executados sem efeitos colaterais nocivos para outros processadores (ou

[13] *Software transactional memory.*

para o sistema no seu todo). Por exemplo, um serviço que invoque uma rotina de escrita para um dispositivo externo (ou para o caso, para qualquer ficheiro externo), ou que receba informação de entidades externas ao programa, não pode, evidentemente, ser repetida transparentemente. Por outro lado, serviços que apenas modificam atributos do objecto são repetíveis.

Serviços repetíveis

Um serviço será repetível se o seu efeito no estado do sistema – programa e eventuais entidades externas que interagem com o serviço – como resultado da sua execução, é descartável como se o serviço nunca tivesse executado.

Assim, este esquema de sincronismo é realizável estaticamente de uma forma segura se o sistema de compilação for capaz de identificar correctamente todos os serviços repetíveis de cada objecto concorrente (não permitindo a sua escolha casa algum dos serviços não seja repetível).

Uma vez mais, chama-se a atenção de que, ao contrário dos esquemas de sincronismo previamente apresentados, o sincronismo sem bloqueamento não estão ainda integrados, e devidamente experimentos, na linguagem protótipo que está a ser desenvolvida (espera-se que esta situação mude no futuro).

No apêndice C.1, secção C.2 apresenta-se, apenas a título indicativo, uma primeira aproximação para implementar este esquema de sincronismo.

	Monitores	Exclusão Leitores-Escritor	Leitores-Escritor Concorrentes	Sem Bloqueamento
Identificação objectos concorrentes	**Sim**	**Sim**	**Sim**	**Sim**
Identificação consultas puras	**Não**	**Sim**	**Sim**	**Sim**
Identificação consultas puras repetíveis	**Não**	**Não**	**Sim**	**Sim**
Identificação serviços repetíveis	**Não**	**Não**	**Não**	**Sim**

Tabela 5.1: Requisitos colocados por esquemas de sincronismo simples.

5.10.7 Esquemas mistos de sincronismo

A tabela 5.1 sumariza os requisitos mais importantes colocados sobre o sistema de compilação dos quatro esquemas de sincronismo apresentados. Como facilmente se constata, os esquemas que têm um valor médio maior de *COA* são também os que mais requisitos impõem ao sistema de compilação.

No entanto, não existe nenhuma razão, teórica ou prática, para se utilizar um único esquema uniforme para a sincronização de objectos concorrentes. Pode-se considerar também a possibilidade de se utilizar diferentes esquemas de sincronismo, simultaneamente ou alternadamente no tempo, no mesmo objecto concorrente. Abre-se dessa forma a possibilidade, entre outras coisas, de optimizar, de uma forma adaptada a cada objecto, a sua disponibilidade concorrente.

Tal como com os esquemas de sincronismo simples, a escolha de um esquema misto requer a verificação de todas as condições de correcção incluindo, em particular, a necessidade de cobertura total do objecto (secção 5.10.2).

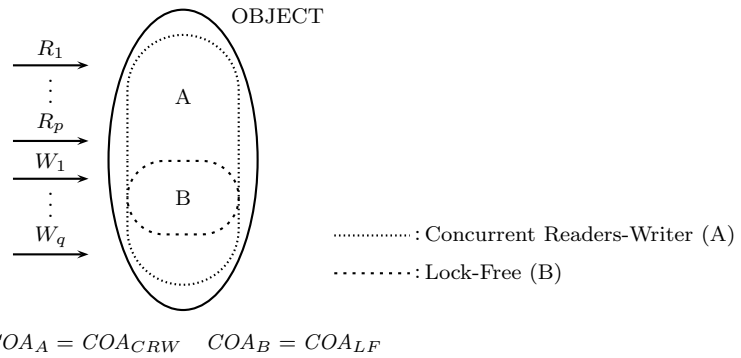

Figura 5.12: Exemplo de um esquema misto de sincronismo.

5.10.8 Esquemas mistos de sincronismo por exclusão mútua

Uma forma possível de combinar vários esquemas de sincronismo num objecto é impor a sua exclusão mútua. Ou seja, deixar que apenas um esteja activo em cada instante. Por exemplo, um objecto pode ter um grupo de serviços sincronizáveis por métodos sem bloqueamento entre eles, e outros que, não sendo repetíveis, requerem exclusão mútua, exclusão leitores-escritor ou leitores-escritor concorrentes, com todos os restantes serviços do objecto (figura 5.12). Para estes casos seria perfeitamente seguro o uso de um mecanismo de exclusão mútua assíncrona de grupos[14] [Joung 00], em que vários processadores poderiam concorrentemente aceder aos serviços com sincronismo sem bloqueamento, em exclusão mútua com processadores a tentar aceder aos outros serviços do objecto. Em tempo de execução, o objecto concorrente alternaria (podendo-se impor, ou não, diferentes prioridades), consoante as necessidades, entre os vários sub-esquemas de sincronismo.

Outra situação com uma solução similar ocorre quando há o interesse de um objecto ter um sincronismo diferente consoante o uso e o contexto onde é utilizado. Por exemplo, pode haver a necessidade de reservar o uso exclusivo de um objecto para uma sequência de chamadas aos seus serviços[15]. Se esse objecto tiver por omissão um sincronismo sem bloqueamento, e se esta situação não for acautelada, não seria possível implementar este tipo de uso exclusivo do objecto, limitando a usabilidade do sincronismo sem bloqueamento. Uma solução para este problema será implementar os dois tipos de sincronismo (sem bloqueamento e exclusão leitores-escritor), recorrendo novamente ao mecanismo de exclusão mútua assíncrona de grupos para impedir o uso simultâneo dos dois tipos de sincronismo (que não podem, em caso algum, ser aplicados simultaneamente ao mesmo grupo de serviços de objectos). Desta forma consegue-se um uso dinâmico seguro de diferentes tipos de sincronismo nos mesmos objectos, aproveitando ao máximo os mecanismos menos restritivos em termos de concorrência intra-objecto.

[14]Curiosamente, a autor pensou e desenvolveu uma classe para implementar este esquema de sincronismo (secção D.11) antes de constatar que já existia uma publicação que o descrevia.
[15]Este caso é tratado com mais detalhe na secção 5.12.

Correcção na mistura de sincronismos por exclusão mútua

É seguro o uso e qualquer combinação de esquemas mistos em exclusão mútua se as seguintes condições forem observadas:

a) Existir cobertura total do objecto;

b) Cada um dos sub-esquemas de sincronização for seguro relativamente ao conjunto de serviços do objecto a que se aplica (que será um subconjunto de todos os serviços do objecto).

A demonstração deste critério de correcção é imediata. Uma vez que o mecanismo de exclusão mútua assíncrona de grupos, por definição, garante que no máximo apenas um dos sub-esquemas de sincronização está activo, e sendo também garantido que todos os serviços do objecto são sincronizados por pelo menos um dos tipos de sincronismo (podem estar sujeitos a mais do que um, embora, como é evidente, não simultaneamente), facilmente se conclui que é suficiente garantir que cada um dos sincronismos seja seguro relativamente ao subconjunto de serviços do objecto aos quais se aplica.

5.10.9 Esquemas mistos de sincronismo em concorrência

Por definição, a larga maioria das combinações em concorrência de esquemas de sincronismo não são seguras. A modificação concorrente de atributos de um objecto leva quase sempre a problemas de competição dessincronizada sobre esses recursos, dos quais podem resultar, de uma forma não previsível, valores sem sentido para esses atributos, quebrando o invariante da classe.

No entanto, em certas situações muito particulares parece poder fazer sentido permitir disciplinadamente o acesso concorrente ao objecto, mesmo sem que para tal se obrigue a um sincronismo sem bloqueamento, ou de concorrência leitores-escritor. Por exemplo, a utilização em concorrência de duas, ou mais, zonas de exclusão mútua ou de leitores-escritor (figura 5.13), dentro de um objecto – cada uma delas protegendo um grupo distinto de atributos – não sendo em geral segura uma vez que nada garante que nessa situação o invariante se verificará – pode, desde que impostas algumas restrições no seu uso, ser linearizável.

Utilizando uma analogia com um exemplo real, caso tivéssemos um objecto do tipo `CARRO` poder-se-ia de uma forma segura substituir um pneu em concorrência com a afinação do motor, isto mesmo sem sermos obrigados à utilização de um sincronismo sem bloqueamento (ou seja, sem a necessidade de exigir que ambas as operação sejam repetíveis).

A execução de um serviço de um objecto será correcta se o critério de condição de serviços se verificar (página 27). Assim sendo, e assumindo apenas chamadas a serviços do objecto que podem modificar o seu estado (em geral: comandos), a execução apresentada na figura 5.14 não é correcta, uma vez que o processador P_1 não pode testar

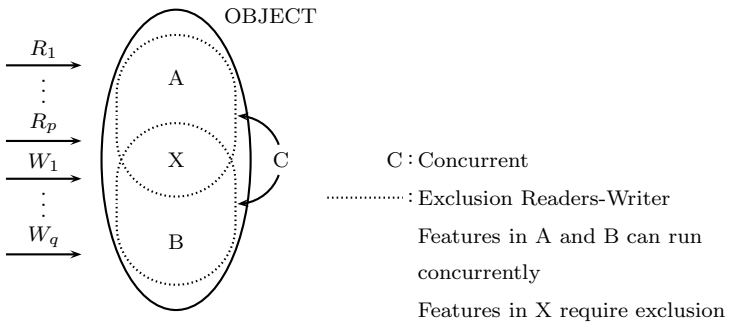

Figura 5.13: Dupla exclusão leitores-escritor.

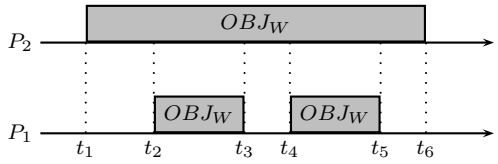

Figura 5.14: Execução errada num objecto com mistura de sincronismo em concorrência.

o invariante de uma forma segura no intervalo $[t_3, t_4]$ entre duas chamadas a serviços do objecto.

Verificação linearizável de invariantes

Analisando a figura 5.14 podem-se fazer algumas constatações. Do ponto de vista do processador P_1 seria linearizável antecipar a verificação do invariante do instante t_2 para o instante t_1, uma vez que, se só existisse o processador P_1 a executar o objecto, caso o invariante se verifique em t_1 também necessariamente se verificará em t_2. Será assim, perfeitamente aceitável reutilizar o teste ao invariante feito por P_2 em t_1, para o processador P_1 em t_2 (ou seja, aceitar o resultado do teste ao invariante em t_1).

Da mesma forma será linearizável atrasar e reutilizar o teste do invariante de P_1 em t_3 para P_2 em t_6, desde que não seja permitida entretanto mais nenhuma chamada ao objecto por parte de P_1 (figura 5.15). Mais, no caso de se querer gerar excepções correctamente, o processador P_1 terá que ser bloqueado até o instante t_6, uma vez que só nessa altura é que o invariante da classe pode ser testado (podendo este falhar, no caso de o programa ter erros, e podendo essa falha resultar da execução de um qualquer

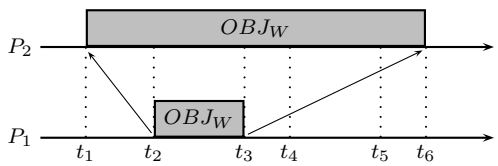

Figura 5.15: Execução correcta num objecto com mistura de sincronismo em concorrência.

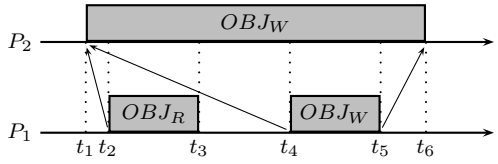

Figura 5.16: Execução correcta num objecto com mistura de sincronismo em concorrência.

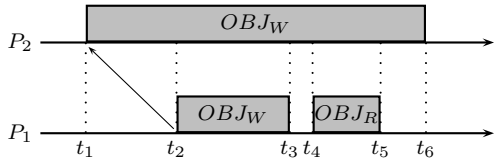

Figura 5.17: Execução errada num objecto com mistura de sincronismo em concorrência.

dos dois processadores).

Já a situação apresentada na figura 5.16, apesar de envolver duas invocações por parte do processador P_1 em concorrência com um única de P_2, é passível de ser considerada segura, uma vez que o invariante não é alterado durante a execução de consultas puras do objecto, pelo que o invariante verificado em t_1 pode ser reutilizado em t_2, t_3 e t_4.

O caso apresentado na figura 5.17 não é correcto uma vez que aquando do início da execução em t_4 por P_1 do serviço de leitura sobre o objecto, não é possível reutilizar nem verificar o invariante.

Para completar a análise a este tipo de sincronismo falta ainda abordar duas situações. A primeira ocorre quando a primeira execução em concorrência sobre o objecto é feita num serviço de leitura. Neste caso, facilmente se constata que o invariante verificado no início desse serviço se pode reutilizar directamente para outros serviços que posteriormente sejam executados em concorrência (uma vez que, por definição, os serviços de leitura não modificam o invariante da classe).

Por fim, nada impede que o último serviço de escrita a ser feito em concorrência sobre o objecto tenha de ser o primeiro que iniciou essa zona de concorrência (como acontece nas figuras apresentadas). O que se impõe é que o invariante de entrada seja o existente no início da execução do primeiro processador escritor e que o invariante de saída seja o que ocorre no fim da execução do último processador escritor.

Generalizando todos estes casos:

Verificação concorrente de invariantes

Na execução concorrente de vários processadores num objecto na presença de esquemas mistos de sincronismo em concorrência, é linearizável verificar o invariante apenas quando o primeiro processador escritor inicia a execução no objecto, e quando o último processador escritor termina, se nesse intervalo de tempo as seguintes condições se verificarem:

a) Cada processador executa, no máximo, um único serviço de escrita sobre o objecto;

b) Cada processador executa zero ou mais serviços de leitura desde que obrigatoriamente precedam a execução do eventual serviço de escrita no mesmo processador.

Voltando ao exemplo do carro, com um esquema de sincronismo concorrente com múltiplas zonas de exclusão leitores-escritor respeitando este critério, teríamos a possibilidade de simultaneamente afinar o motor e trocar pneus por diferentes funcionários (processadores), mas com a restrição de cada funcionário apenas poder realizar uma operação por cada operação realizada por todos os outros funcionários. Ou seja, cada funcionário só poderá prosseguir o seu trabalho com a garantia de o anterior ter sido feito correctamente (a existir pós-condição na respectiva operação) não comprometendo a correcção do estado do carro (expressa pelo respectivo invariante). Não é difícil constatar que todas estas considerações são igualmente aplicáveis à mistura por concorrência de outros tipos de sincronismo.

Correcção na mistura de sincronismos com concorrência

É seguro misturar em concorrência dois ou mais mecanismos de sincronismo desde que se verifiquem as seguintes condições:

a) Cobertura total do objecto;

b) Cada mecanismo de sincronismo protege um diferente grupo de atributos do objecto;

c) O critério de verificação concorrente de invariantes é satisfeito.

Realizabilidade

Uma característica interessante dos esquemas mistos de sincronismo é o facto de as exigências colocadas por cada sub-esquema não terem necessariamente de se aplicar a

todo o objecto, mas apenas a um subconjunto deste.

Para a verificação automática da realizabilidade dos esquemas mistos de sincronismo em concorrência é necessário que o sistema de compilação associe a cada serviço o conjunto de atributos que podem ser modificados (directa ou indirectamente). Só serviços que nunca interfiram entre si podem ser executados concorrentemente

Para implementar um algoritmo de sincronização para este esquema é suficiente utilizar uma aproximação simples baseada num contador atómico partilhado. Na secção C.3 é mostrada uma possível implementação segura (em C) da verificação do invariante para objectos com este sincronismo no caso em que os processadores são POSIX-THREADS. Nesta implementação toda a sincronização necessária é feita na verificação do invariante.

5.10.10 Escolha dos esquemas de sincronismo

Apresentados os vários esquemas seguros de sincronismo passíveis de realizações automáticas pelo sistema de compilação é necessário agora tratar o problema de se expressar a sua escolha em programas concorrentes.

Escolhas pré-definidas na linguagem

Esta opção é de longe a mais frequente. É a seguida, por exemplo, pela linguagem ADA95 em que os objectos partilhados (*protected types*) são sincronizados de uma forma segura com o mecanismo de sincronismo de exclusão leitores-escritor [Ada95 95].

Outra hipótese mais flexível será definir na linguagem diferentes anotações (no sistema de tipos) para diferentes esquemas de sincronismo, deixando para o programador a escolha do esquema desejado para cada objecto.

```
-- synchronization keywords:
--   monitor, exrw, crw, lockfree



-- class declaration definition:
shared monitor class SHARED_OBJECT
...
end


shared exrw class SHARED_OBJECT
...
end


shared crw class SHARED_OBJECT
...
end
```

```
-- mixed synchronization scheme:
shared class SHARED_OBJECT
feature lockfree
...
feature exrw
...
end


-- entity declaration definition:
class SOME_CLASS
...
feature
  a_procedure is
    local
      obj: shared crw OBJECT;
    do
      ...
    end
...
end
```

Figura 5.18: Exemplo de escolha directa do esquema sincronismo.

Na figura 5.18 apresenta-se uma aproximação (em PSEUDO-EIFFEL) em que, para além da indicação de partilha de cada objecto (**shared**), é incluída uma anotação referente à escolha do sincronismo desejado: **monitor** para indicar exclusão mútua; **exrw** para exclusão leitores-escritor; **crw** para leitores-escritor concorrentes e **lockfree** para um sincronismo sem bloqueamento.

Esta aproximação é simples e faz com que a associação entre os esquemas de sincronismo e os objectos partilhados seja directa e evidente. No entanto, ela vai contra um dos objectivos estabelecidos neste trabalho – a sincronização abstracta de objectos (secção 5.9.1) – pelo que não será uma opção a considerar.

Escolha automática pelo sistema de compilação

Uma opção no sentido da sincronização abstracta é delegar integralmente a escolha dos esquemas de sincronismo considerados mais apropriados no sistema de compilação. Para essa escolha, o sistema de compilação pode fazer uso de heurísticas apropriadas. Por exemplo, caso seja identificada a possibilidade de o sincronismo intra-objecto de um objecto concorrente poder gerar *deadlocks*, o sistema de compilação pode optar por utilizar, caso seja possível, um esquema de sincronismo sem bloqueamento resolvendo, dessa forma, esse problema.

No entanto, esta opção poderá ser pouco flexível, já que não permite que o programador possa ter uma palavra a dizer nessa escolha (para mais sabendo-se que não existem heurísticas óptimas para todas as aplicações possíveis de objectos concorrentes).

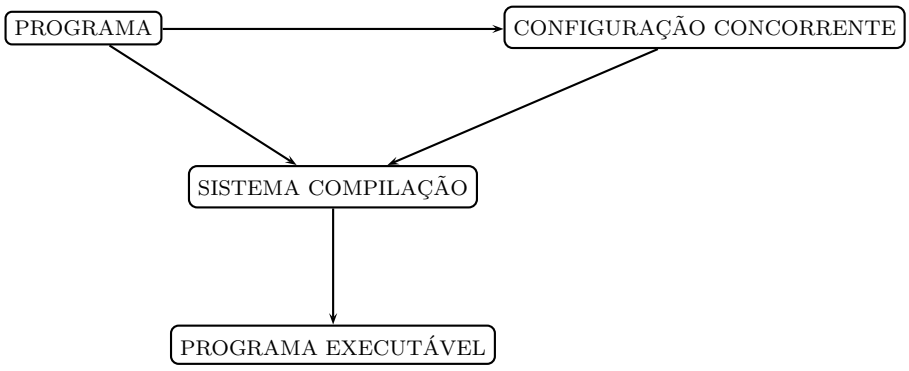

Figura 5.19: Esquema da escolha partilhada de sincronismo.

Escolha partilhada

Uma terceira possibilidade consiste em partilhar a escolha entre o sistema de compilação e o programador. Esta será a aproximação ideal desde que o sistema de compilação não permita que o programador escolha esquemas inseguros, mas que, ao mesmo tempo, dê liberdade para a escolha de um qualquer esquema seguro. Temos assim a junção do melhor dos dois mundos: a segurança da escolha estática do sincronismo e a flexibilidade da escolha pelo programador do sincronismo mais apropriado para cada objecto.

Para que esta coexistência seja possível é desejável que as eventuais escolhas do programador não sejam feitas directamente dentro do programa, mas sim numa especificação separada recorrendo, por exemplo, a uma linguagem externa de configuração e especificação concorrente do programa.

A figura 5.19 esquematiza esta aproximação. A configuração concorrente faz uso do código fonte do programa para identificar sem ambiguidades os objectos concorrentes para os quais se pretende escolher um determinado sincronismo. Por sua vez, o sistema de compilação necessita quer do programa (obviamente) quer da configuração concorrente para estaticamente verificar se as escolhas feitas são possíveis, e se assim for, gerar o programa executável.

Na linguagem protótipo que tem vindo a ser desenvolvida neste trabalho, pretende-se que a configuração concorrente seja feita recorrendo a uma linguagem de controlo de concorrência. Uma apresentação dessa linguagem pode ser consultada no capítulo 6, secção 6.7.

5.11 Sincronização condicional

No contexto das linguagens orientadas por objectos puras, e assumindo uma estratégia de espera (secção 4.6.3), a sincronização condicional é um mecanismo, com

eventual bloqueamento, de acesso condicional exclusivo a objectos.

A necessidade deste sincronismo pode resultar exclusivamente de uma condição interna ao objecto ou, em alternativa, de condições externas impostas por clientes desse objecto. No primeiro caso a sincronização condicional aplica-se sobre o sincronismo intra-objecto, e no segundo sobre o sincronismo inter-objecto.

Ambos os modelos de comunicação entre processadores – envio de mensagens ou partilha de objectos – podem também requerer mecanismos de sincronização condicional. Independentemente do modelo, caso a comunicação seja síncrona (secção 4.5.1) este sincronismo vai impor um bloqueamento no processador que requer a execução (condicional) de um serviço da classe. Já no caso da comunicação assíncrona, a espera dá-se na fila das mensagens por tratar associada ao objecto (ou ao processador).

Nesta secção aborda-se apenas o problema da realizabilidade automática deste sincronismo. O problema da escolha dos mecanismos de linguagem que o podem expressar será tratado na secção 5.14.

5.11.1 Comunicação síncrona

Na implementação do sincronismo condicional para mecanismos síncronos de comunicação entre processadores pode-se fazer uma aproximação similar à utilizada em monitores [Hoare 74]. Os monitores utilizam, para esse fim, as chamadas variáveis de condição. Estas variáveis, às quais não está associado nenhum valor, são abstracções para filas de espera de processadores, podendo-se-lhes aplicar três operações[16]: espera (*wait*), sinaliza (*signal*) e sinaliza todos (*broadcast*). O efeito dessas operações é o seguinte. A operação de espera faz com que o processador que a requer seja colocado na fila de espera associada à variável de condição (libertando o monitor para outros processadores); a operação sinaliza faz com que um dos processadores seja retirado da fila sendo-lhe dado, assim que possível, o acesso exclusivo ao monitor; por fim, a operação sinaliza todos faz o mesmo que a operação anterior mas para todos os processadores existentes na fila de espera. A biblioteca POSIX-Threads para a linguagem C implementa este tipo de variáveis.

Esta aproximação ao sincronismo condicional tem, no entanto, um grave problema: ela não é estaticamente segura já que delega nas mãos dos programadores a responsabilidade de as declarar e utilizar correctamente as variáveis de condição. Para além de não ser segura, também não é suficientemente abstracta, já que o programador é obrigado a construir o código de sincronização condicional ligando-o explicitamente às verdadeiras condições associadas ao estado dos objectos (aproximação operacional). O próprio Hoare [Hoare 74, página 556] reconhece que uma aproximação alternativa assente em instruções de espera condicional seria mais simples e segura. Por outro lado, esta aproximação permite a implementação de algoritmos de sincronismo bastante eficientes já que o programador tem a possibilidade de decidir quais os pontos do programa em que é necessário sinalizar processadores e, mais importante ainda, pode decidir para que processadores esses sinais serão endereçados (recorrendo a diferentes variáveis de condição).

[16] Na proposta inicial de Hoare (e Brinch Hansen) [Hoare 74] eram só duas operações: *wait* e *signal*.

Na linguagem JAVA a responsabilidade para gerir os mecanismos (de base) de sincronização condicional (designados por *Wait*, *Notify* e *NotifyAll*) pertence também ao programador. No entanto, ao contrário dos monitores originais, não existe a possibilidade de declarar várias variáveis de condição por objecto, e como tal, de escolher diferentes grupos de processadores (que em JAVA são *threads*) nas operações de sinalização (notificação). Em JAVA existe uma única variável de condição por objecto, à qual se aplica as operações de espera e notificação. Assim, um sinal de notificação acorda um qualquer processador presente na fila de espera, independentemente da condição de espera que lhe esteja associada. Se houver vários processadores à espera de diferentes condições de sincronização, existe a possibilidade de uma notificação acordar o processador errado (situação que aconselha o uso alternativo de notificações para todos [Lea 00, páginas 191–192]).

No entanto, nenhuma destas aproximações se aproxima dos objectivos pretendidos: sincronização segura, abstracta e automaticamente realizável pelo sistema de compilação da linguagem.

Um algoritmo possível nesse sentido[17] será associar uma única variável de condição a cada objecto (como em JAVA), implementando todas as acções de espera condicional como operações de espera nessa variável (sejam as relacionadas com o sincronismo intra-objecto, ou com o sincronismo inter-objecto), e colocando operações de sinalização para todos os processadores sobre essa variável no fim de todas as rotinas públicas do objecto[18]. Os processadores, ao ganharem o acesso exclusivo ao objecto, verificam se a condição que os fez esperar (se existir alguma) é verdadeira, executando a rotina caso o seja, ou voltando a colocar-se em espera sobre a variável de condição caso não o seja. Obviamente que este algoritmo, apesar de ir de encontro aos objectivos pretendidos, é potencialmente muito ineficiente.

Este algoritmo pode ser melhorado caso o sistema de compilação tenha a capacidade de distinguir entre comandos e consultas (puras). Nesta situação só é necessário sinalizar todos os processadores em espera no fim da execução de comandos (e de eventuais consultas não puras), uma vez que apenas estas rotinas podem alterar as condições de espera.

Esta é a implementação automática utilizada neste momento na linguagem protótipo que está a ser desenvolvida no âmbito deste trabalho (MP-EIFFEL) [OeS 06a]. Os exemplos de implementação automática dos vários esquemas de sincronização intra-objectos apresentados na secção C.1 utilizam também este algoritmo.

Possíveis implementações mais eficientes

Neste problema da implementação do sincronismo condicional, a aproximação operacional – na qual são os programadores que a implementam directamente – apesar da sua insegurança (estática) é ainda a que melhor consegue construir algoritmos muito eficientes.

[17] Similar ao apresentado por Hoare [Hoare 74, página 557] na descrição da implementação de instruções de espera condicional.

[18] Uma vez mais podemos constatar a importância de impor a inexistência de atributos publicamente modificáveis, já que, nessa situação, a sinalização dos processadores em fila de espera poderia ter de ser propagada para todos os clientes da classe que pudessem modificar atributos públicos.

Em [OeS 06a] propomos duas aproximações (mas que carecem ainda de implementação e validação experimental) que podem fornecer algoritmos seguros para este problema e que se aproximam bastante mais da eficiência dos algoritmos feitos directamente pelos programadores. Uma das aproximações faz uso das asserções concorrentes, e a outra da associação entre todas as rotinas da classe aos atributos dos quais dependem, ou modificam.

5.11.2 Comunicação assíncrona

Numa comunicação assíncrona entre processadores não há bloqueamento de processadores como consequência deste aspecto de sincronismo. A espera faz-se antes na fila de mensagens a tratar pelo processador receptor. Nesta situação o processador receptor só retirará a mensagem da fila caso a condição de espera se verifique. Caso contrário, passará à mensagem seguinte (desde que, para não comprometer a consistência sequencial (página 65), não tenha tido origem no mesmo processador). No fim do processamento de cada mensagem recebida, e de preferência antes mesmo de passar à próxima mensagem na fila, o processador receptor terá de verificar se existe alguma mensagem anterior em espera condicional e, caso a condição de espera seja verdadeira, executá-la.

5.12 Sincronização inter-objecto

A realização (automática) deste sincronismo requer a utilização de algoritmos de reserva exclusiva de objectos. Esses algoritmos dependem do modelo de comunicação a utilizar.

Tal como aconteceu no caso do sincronismo condicional, nesta secção vamos abordar apenas o problema da realizabilidade automática deste sincronismo. A sua integração em linguagens concorrentes será tratada na secção 5.15.

5.12.1 Comunicação por envio de mensagens

Em mecanismos de comunicação (entre processadores) por envio de mensagens é necessário poder reservar objectos remotos para responderem apenas a mensagens com origem, directa ou indirecta, no processador onde essa sincronização é requerida.

Com este modelo de comunicação é necessário prever a situação em que um processador possa ter de responder a mensagens de outros processadores que não o que fez a reserva exclusiva dos objectos, devido a este último processador lhes ter delegado essa responsabilidade. Por exemplo, vamos supor que temos três processadores: *P1*, *P2* e *P3*, cada um deles a gerir mensagens enviadas, respectivamente, para os objectos: `o1`, `o2` e `o3`. Se parte do programa em `o1` tiver a seguinte invocação remota: `o2.do_something(o3)`, então caso *P1* não passe temporariamente a reserva para *P2* teremos como provável resultado o programa ficar eternamente bloqueado (*deadlock*). Estes problemas de passagem de testemunho no sincronismo inter-objecto, no contexto da proposta SCOOP, estão tratados em [Nienaltowski 06a].

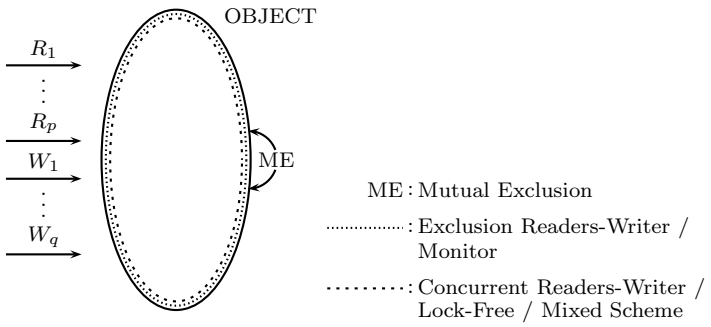

Figura 5.20: Esquema misto de sincronismo para reserva de objectos.

5.12.2 Comunicação por partilha de objectos

A implementação automática neste modelo de comunicação faz-se recorrendo a um esquema de exclusão mútua tipo monitor (secção 5.10.3). Como a causa para este sincronismo é externa ao objecto, muito embora a sua implementação possa residir no próprio objecto (como veremos), não se poderá utilizar o mecanismo de sincronismo intra-objecto (mesmo que ele seja um monitor) também para este fim. Quer isto dizer que os objectos concorrentes poderão ter a si associados dois esquemas de sincronismo: um para garantir segurança intra-objecto e outro para garantir a reserva inter-objecto.

5.12.3 Integração com o sincronismo intra-objecto

Esta situação levanta, obviamente, o problema da realizabilidade automática desta integração de mecanismos de sincronismo, sendo que um deles – o intra-objecto – pode até ser em esquema de sincronismo sem bloqueamento.

Uma solução simples e elegante para este problema assenta no esquema de sincronismo misto por exclusão mútua (secção 5.10.8). A figura 5.20 mostra como essa integração funciona. O sincronismo intra-objecto (seja qual for o esquema utilizado) pertence a um grupo, e o sincronismo inter-objecto pertence a outro. Assim, não é possível aparecerem interferências inseguras entre ambos. Por outro lado, podem aparecer problemas de *liveness*, que não serão tratados no presente trabalho.

Um aspecto interessante da realização automática proposta para o sincronismo inter-objecto é o facto de a sincronização, apesar de ser requerida externamente, residir no próprio objecto, o que facilita tremendamente a sua implementação prática.

É importante referir-se que uma aproximação a este problema assente em zonas de exclusão mútua (*mutex*) recursivas, como é incentivado em Java, não é aceitável. Não só por restringir o sincronismo intra-objecto a um monitor (o que seria por si só suficiente para a não considerar), mas também porque é insegura ao não separar claramente os dois aspectos de sincronismo.

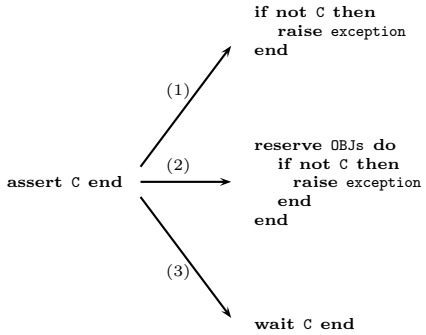

Figura 5.21: Comportamentos possíveis na presença de asserções concorrentes.

5.13 Outros mecanismos orientados por objectos em concorrência

Um dos aspectos mais complexos quando se pretende estender linguagens orientadas por objectos com mecanismos concorrentes consiste nas possíveis interacções destes com os mecanismos orientados por objectos. Algumas dessas interacções podem ser potencialmente inseguras, pelo que é necessário encontrar soluções que evitem esses problemas. Outras, pelo contrário, abrem a possibilidade altamente desejável de se poderem definir comportamentos sinérgicos quando utilizadas em concorrência.

As próximas secções vão estudar os problemas de segurança e as possibilidades de sinergias para alguns dos mecanismos orientados por objectos apresentados no capítulo 3 no contexto de linguagens concorrentes.

5.14 Asserções concorrentes

Como deve um programa comportar-se na presença de asserções concorrentes[19]?

A figura 5.21 mostra as três respostas possíveis. Como, por definição, uma asserção concorrente depende pelo menos de outro processador que não o processador que está a testar a asserção, o comportamento sequencial não sincronizado – comportamento `(1)` na figura – criaria claramente uma competição dessincronizada pela verificação da condição `C`, sendo por isso uma opção insegura e inaceitável.

Outra possibilidade – designada por `(2)` na figura – seria reservar incondicionalmente todos os objectos concorrentes envolvidos na asserção, testando-a posteriormente como se fosse uma asserção sequencial. Este comportamento é também uma potencial fonte de uma competição dessincronizada, embora menos crítica que a anterior. Como a reserva exclusiva dos objectos concorrentes não depende da condição existente na asserção, a não ser que essa condição seja garantida pelos invariantes desses objectos, essa condição pode ser verdadeira ou falsa dependendo apenas da altura em que ocorre essa reserva (ou seja da velocidade relativa dos processadores envolvidos). Nesta situação a asserção deixaria, pura e simplesmente, de poder ser utilizada como um teste de correcção, perdendo a sua utilidade.

[19] Secção 5.1.3.

A última possibilidade consiste em associar as asserções concorrentes a esperas condicionais: uma asserção concorrente faz com que o processador que a está a testar espere até que ela se verifique [OeS 06a]. A proposta SCOOP desde o início que associava esse comportamento às pré-condições separadas [Meyer 97, página 993], mas só muito recentemente é que se propôs que esse comportamento fosse estendido a outras asserções [Nienaltowski 06b][20].

Podemos encarar as asserções como sendo condições de correcção que se aplicam aos excertos do programa existentes a montante da sua localização. É sempre da responsabilidade desse código garantir que essas asserções se verificam. No caso das pré-condições será da responsabilidade dos clientes, sendo da responsabilidade da própria classe garantir o invariante e as pós-condições. Num programa sequencial só existe um processador pelo que se se verificar (geralmente testando-a em tempo de execução) que uma asserção é falsa então estamos inequivocamente na presença de um erro no programa (todas as acções no programa só podem ser executadas por esse processador). No entanto, os programas concorrentes podem ter mais do que um processador, pelo que se uma asserção é concorrente há a possibilidade do seu valor poder variar independentemente do programa do processador que a verifica em tempo de execução. Temos assim que a responsabilidade de garantir essa asserção não pertence necessariamente ao processador que a está a verificar, mas eventualmente a outros processadores. Continua assim a ser um critério de correcção, mas não aplicável necessariamente ao processador que verifica a asserção, pelo que novamente se conclui que o único comportamento seguro é fazer com que essas asserções sejam instruções de espera condicional.

Temos assim uma sinergia muito interessante entre os mecanismos de suporte à execução de contratos e a sincronização condicional.

A espera condicional, no entanto, não é suficiente para garantir a validade de algumas das asserções como é o caso das pré-condições concorrentes. As pré-condições servem para garantir a verificação de uma condição no início da rotina à qual estão ligadas. Ou seja, para que uma pré-condição faça sentido é necessário que entre a sua verificação e a execução do corpo da rotina, a condição se mantenha. Logo, para além da eventual espera condicional, é necessário também garantir que os objectos concorrentes associados à pré-condição concorrente estejam reservados para uso exclusivo nessa rotina. Ou seja, nesta situação é necessário impor um sincronismo inter-objecto condicional a esses objectos. O mesmo acontece com o invariante da classe no início da rotina, mas não com as pós-condições nem com o invariante no final da rotina[21].

5.15 Selecção algorítmica por condições concorrentes

Podemos considerar que as pré-condições de uma rotina, assim como o invariante da classe, seleccionam condicionalmente o programa expresso no corpo dessa rotina, uma vez que só faz sentido executar o corpo da rotina se essas condições se verificarem. Essa é a razão de fundo pela qual é necessário garantir a reserva exclusiva dos objectos

[20] Muito embora a existência de asserções concorrentes em SCOOP esteja limitada a condições utilizando argumentos formais separados.

[21] Este racíocinio aplica-se também às outras asserções algorítmicas.

```
if CONDITION then
  precondition CONDITION do
    COMMANDS
  end
end
```

```
while CONDITION do
  precondition CONDITION do
    COMMANDS
  end
end
```

Figura 5.22: Instruções condicionais e repetitivas estruturadas.

concorrentes que eventualmente lhes estejam associados. Ou seja, uma sincronização inter-objecto.

É muito interessante constatarmos que este raciocínio axiomático não se aplica somente a essas asserções. De facto, o mesmo acontece com as instruções estruturadas puras (página 16) que seleccionem algoritmos por condições concorrentes, como é o caso das instruções condicional e repetitiva. A figura 5.22 apresenta o comportamento axiomático que é esperado nestas duas instruções[22].

Assim estas instruções estruturadas puras só serão seguras se, também neste caso, se impuser a reserva exclusiva (aplicável durante todo o bloco da instrução) dos eventuais objectos concorrentes envolvidos nas condições lá expressas.

Outro aspecto semântico muito interessante em todos estes efeitos sinérgicos é o facto de, ao contrário das asserções concorrentes, não fazer qualquer sentido associar uma acção de espera condicional a estas instruções estruturadas puras. De facto, as condições concorrentes eventualmente envolvidas nestas instruções não são condições de correcção (mas sim de selecção algorítmica), pelo que ambos os valores possíveis da condição são essenciais para a correcção do algoritmo.

Assim o comportamento de reserva exclusiva de objectos não deve ser confundido com o de espera condicional requerido nas asserções concorrentes. Isto apesar de no caso das pré-condições ambos os comportamentos lhes estarem associados.

Com esta semântica associada às condições concorrentes[23] consegue-se simultaneamente, não só garantir a segurança e melhorar a expressividade da linguagem, como também optimizar a disponibilidade concorrente dos objectos, já que a reserva exclusiva de objectos concorrentes será feita só quando é estritamente necessária.

Existem, é claro, outras formas de se expressar o sincronismo inter-objecto. Uma delas consiste no uso da instrução estruturada apresentada na secção 4.6.4. Outra hipótese é a utilizada em SCOOP (secção A.6). No entanto nenhuma destas aproximações (ou outras quaisquer), para ser segura, evita a necessidade de garantir a reserva exclusiva de objectos na utilização de pré-condições concorrentes e nas instruções de selecção e repetição que façam uso de condições concorrentes.

5.16 Herança (relação subclasse)

As interferências entre o mecanismo de herança (subclasse) e o código de sincronismo de objectos concorrentes tem sido uma das áreas mais estudadas e que mais problemas

[22] Omitimos a instrução repetitiva **repeat**... **until** porque ela converte-se de uma forma trivial numa instrução repetitiva do tipo **while**.

[23] Para as quais temos um artigo em desenvolvimento a ser submetido para publicação [OeS 06b].

tem trazido na integração de mecanismos de concorrência em linguagens orientadas por objectos [America 87a, Briot 87, Kafura 89, Matsuoka 93]. Os problemas identificados prendem-se, basicamente, com a dificuldade em reutilizar o código de sincronismo, obrigando a que este seja redefinido, parcial ou mesmo totalmente. Estes problemas foram designados por *anomalias de herança* [Matsuoka 93], existindo inúmeras propostas para as resolver [Matsuoka 93, McHale 94, Baquero 95, Holmes 99, Lu 01].

Apesar do número muito elevado de publicações referindo-se directamente a estas anomalias de herança, a maioria não define com precisão esse termo. Holmes [Holmes 99, página 43], reconhecendo essa dificuldade, propõe uma definição:

> Considere uma linguagem orientada por objectos com um mecanismo de herança em particular e notações para fornecer concorrência e sincronização. Se utilizarmos a herança sobre uma classe base e descobrirmos que a introdução de novos métodos obriga à redefinição dos métodos da classe base ou da respectiva sincronização, então estamos perante um problema de anomalia de herança.

Em aproximações que fazem uma abordagem explícita ao sincronismo (página 55) é natural que surjam anomalias de herança. Sendo da responsabilidade directa do programador a construção de um algoritmo de sincronismo correcto, esse algoritmo tende a estar fortemente ligado à classe para a qual é feito, podendo não se adaptar devidamente ao aparecimento de novos serviços ou a redefinições de serviços existentes em subclasses. Essa ligação forte dificulta também a modificação do esquema de sincronismo em subclasses.

Por outro lado, uma aproximação implícita ao sincronismo, como é estudada e proposta neste trabalho, tende a ser imune a esses problemas já que a implementação adequada do sincronismo é feita automaticamente pelo sistema de compilação. A sincronização abstracta evita, em grande medida, que esta opção represente uma perda no controlo e ajuste do sincronismo dos objectos concorrentes.

5.17 Polimorfismo de subtipo

Aproximações à concorrência orientadas por objectos que não façam uso do sistema de tipos para identificar as entidades com tipo associadas a objectos concorrentes levantam problemas de segurança. Sendo as relações de subtipo impostas pelo sistema de tipos, nessas condições com facilidade se podem fazer passar objectos sequenciais como se fossem concorrentes ou vice-versa criando, geralmente[24], problemas sérios de utilização concorrente de objectos não sincronizados. Temos assim outra razão muito forte (para além da apresentada na secção 5.2.1) para que se faça uso do sistema de tipos para separar objectos concorrentes dos sequenciais.

[24]Dependendo do modelo de comunicação entre processadores e da implementação do sincronismo feita em cada linguagem.

5.17.1 Modelo de comunicação por envio de mensagens

No modelo de comunicação por envio de mensagens (assumindo um mecanismo de identificação indirecta de processadores como apresentado na secção 5.6.2), não havendo concorrência intra-objecto, os problemas de substitutabilidade colocam-se essencialmente quando se associa um objecto concorrente a uma entidade com tipo sequencial [Meyer 97, página 973]. Nessa situação, o programa (e o sistema de compilação) espera uma comunicação síncrona de e para o mesmo processador e nunca uma comunicação remota potencialmente assíncrona. Já a situação inversa, associar um objecto sequencial a uma entidade com tipo concorrente, poderá não ser tão crítica já que se pode considerar que a comunicação com o mesmo processador é um caso particular (logo substituível) da comunicação genérica de um processador com outro (por exemplo, em SCOOP essa situação é permitida).

5.17.2 Modelo de comunicação por partilha de objectos

Quando passamos para um modelo de comunicação entre processadores por partilha de objectos a situação é a inversa (há uma dualidade entre os dois modelos). Aqui a situação mais insegura é fazer-se passar um objecto sequencial (não sincronizado) onde se espera um concorrente (ou seja num contexto onde podem existir vários processadores a tentar utilizar o objecto). Neste caso, teríamos problemas de competição dessincronizada na utilização do objecto com consequências imprevisíveis no comportamento do programa. A situação inversa de se fazer passar um objecto concorrente onde se espera um sequencial poderá não ser crítica já que a utilização de um objecto partilhado por apenas um processador não levanta problemas de segurança.

5.17.3 Substitutabilidade de esquemas de sincronismo intra-objecto

Um aspecto interessante da abordagem de sincronismo intra-objecto abstracto proposta neste trabalho é a total substitutabilidade entre objectos concorrentes (obviamente, relacionados por subtipo) com sincronismos intra-objecto diferentes. Desde que cada objecto concorrente tenha a si associado um esquema de sincronismo seguro, o objecto, do ponto de vista do seu TDA, comporta-se para o seu exterior da mesma maneira independentemente do esquema de sincronismo utilizado.

5.18 Mecanismo de excepções

As excepções servem, essencialmente, como um mecanismo de sinalização interna de falhas no funcionamento de um programa (secção 3.13). Elas são um mecanismo de comunicação interna, tal como as rotinas, mas com a diferença de interromperem bruscamente a execução normal de programas e de passarem a execução para código específico para lidar com elas.

Em linguagens sequenciais essa comunicação envolve sempre o mesmo processador e os objectos envolvidos só são utilizáveis por ele. Num contexto concorrente a situação pode ser bem diferente. Por um lado, as excepções podem ter de ser entregues a um processador que não o que estava a executar o código que as despoletou. Por outro,

pode acontecer que um objecto partilhado deixe de estar disponível para utilizações concorrentes devido a nele ter ocorrido uma excepção. Estas situações dizem respeito, respectivamente, ao modelo de comunicação entre processadores por envio de mensagens e ao de partilha de objectos.

Neste trabalho estamos interessados em estudar com detalhe mecanismos de excepções intimamente ligados com a programação por contrato, ou seja mecanismos disciplinados de excepções (página 34).

Existem vários trabalhos publicados que analisam mecanismos de excepção em concorrência (por exemplo: [Issarny 01, Xu 95, Mitchell 01]) mas que omitem a relação, essencial na aproximação seguida à programação por objectos, com a programação por contrato.

Recentemente [Arslan 06] foi proposta uma aproximação a este problema no âmbito do SCOOP (modelo de comunicação entre processadores por envio de mensagens). No entanto, a proposta aí feita tem vários problemas como os que apresentámos na página 73. Em 2003 [OeS 04] foi apresentada uma proposta para essa integração, no âmbito da linguagem protótipo MP-Eiffel. Muito embora muitos dos aspectos apresentados nesse artigo se mantenham, a actual proposta difere em alguns aspectos (que clarificaremos mais à frente).

Um mecanismo disciplinado concorrente de excepções deve ter em conta quatro aspectos por nós considerados essenciais:

1. propagação das excepções para o destinatário correcto;
2. disponibilidade concorrente dos objectos após a ocorrência de excepções;
3. recuperação de objectos em tempos instáveis;
4. terminação de processadores.

5.18.1 Propagação para o destinatário correcto

Para que o mecanismo de excepções faça sentido é essencial fazer com que as excepções sejam tratadas, caso o programador assim o queira, na localização correcta, ou seja do lado do responsável pela falha. A programação por contrato (secção 3.12) distribui responsabilidades distintas entre as várias partes de um programa consoante o tipo de asserção envolvido (ver tabela 3.1). Assim, uma falha numa pré-condição é da responsabilidade de quem invocou o serviço. Falhas nas restantes asserções são da responsabilidade (interna) do objecto ao qual o serviço pertence.

Este requisito aplica-se facilmente, por definição, a mecanismos de comunicação entre processadores (entre o processador que invoca um serviço e o processador que o executa) síncronos, como os mecanismos de comunicação por partilha de objectos, ou os mecanismos síncronos de comunicação por mensagens. O problema complica-se, como aliás já foi exposto na página 73, na presença de mecanismos assíncronos de comunicação por mensagens. Nesta situação, para manter a sanidade contratual do mecanismo de excepções, é necessário impor a verificação síncrona da pré-condição (obviamente, apenas a sua eventual parte sequencial). Para as restantes asserções não faz sentido impor uma verificação síncrona (seria tornar síncrona uma comunicação

que se pretendia assíncrona), mas é necessário prever a possibilidade de o objecto não conseguir resolver a causa que levou à ocorrência da excepção (que era da sua responsabilidade), e, por isso mesmo, ter de propagar a excepção a quem lhe requereu a execução do serviço (indicando que não foi possível cumprir a sua parte do contrato). A semântica que nos parece fazer mais sentido consiste em propagar a excepção sincronamente com a próxima tentativa de utilizar o objecto por parte do mesmo processador, independentemente de, entretanto, o objecto ter sido recuperado por outros processadores (secção 5.18.3). Esta semântica difere da proposta feita em [OeS 04].

5.18.2 Disponibilidade concorrente de objectos

O segundo aspecto importante (irrelevante em linguagens sequenciais) tem a ver com a disponibilidade concorrente de objectos nos quais foi gerada uma excepção. Parece claro que no caso da execução nesses objectos ter sido interrompida por uma excepção num tempo instável – e em que o próprio objecto se mostrou incapaz de resolver o problema e eventualmente também incapaz também de repor o seu invariante – não se pode permitir a sua utilização posterior como se nada tivesse acontecido (situação em que teríamos um problema sério de segurança, já que os objectos poderiam ser utilizados sem respeitar os respectivos TDAs).

Obviamente, o problema não se coloca quando falha uma pré-condição. Neste caso, o objecto continua num tempo estável e, como tal, perfeitamente utilizável por qualquer processador (incluindo o responsável pela falha na pré-condição).

No caso de falhas não resolvidas em outras asserções, o objecto terá de passar a estar num estado de indisponibilidade concorrente até à sua eventual recuperação. Qualquer utilização normal posterior do objecto deve resultar no envio síncrono de uma excepção para o cliente (falha de invariante).

5.18.3 Recuperação de objectos

O terceiro aspecto a ter em consideração refere-se à necessidade, que por vezes ocorre, de existir um mecanismo de recuperação de objectos que estejam num estado de indisponibilidade concorrente (este aspecto pode também ter a sua utilidade em linguagens sequenciais). Num contexto concorrente é importante que essa recuperação possa ser feita por outro processador que não necessariamente o que desencadeou a sequência de acções que levaram à falha, já que esse processador pode já não estar em execução (por exemplo, por incapacidade de recuperar da excepção). Essa recuperação terá, obviamente, de passar pela execução de algum serviço da classe (no caso: procedimento) mas tal invocação não pode ser feita normalmente.

Meyer [Meyer 97, páginas 417–418] sustenta que quando uma rotina falha, antes da excepção ser propagada para o cliente, o invariante do objecto *tem* de ser restaurado. No entanto, essa exigência dificilmente pode ser garantida em tempo de execução já que poderia gerar programas com ciclos infinitos. Assim, na prática, é possível que uma rotina passe a excepção ao cliente sem garantir que o objecto a que pertence tem o invariante intacto. Será assim útil permitir que um processador, sempre que receba uma excepção por falha de invariante (e só nesse caso), possa no código para lidar

com excepções[25] invocar directamente um qualquer dos procedimentos de criação do objecto (mas, obviamente, sem recorrer à instrução de criação propriamente dita) antes de voltar a tentar utilizar o objecto[26].

Esta proposta assenta no seguinte raciocínio. De todos os serviços de uma classe, os únicos que não são obrigados a verificar o invariante no início da sua execução são os procedimentos de criação de objectos [Meyer 97, página 370]. Para além do mais, esses procedimentos existem precisamente para inicializar os objectos para um estado onde o invariante se verifica. Pelo que tudo se conjuga sinergicamente para que os procedimentos de criação possam servir também para este propósito muito importante em programas concorrentes (mas que também pode ser útil em programas sequenciais).

É importante voltar a referir que a recuperação de um objecto, apesar de o colocar num estado estável, não impede que uma excepção tenha de ser propagada para o processador que desencadeou as acções que levaram à falha do objecto. Só assim esse processador será devidamente informado da falha de contrato.

5.18.4 Excepções e terminação de processadores

O último problema que nos falta abordar é a relação completa entre excepções e processadores. Em linguagens sequenciais, um programa termina indicando uma falha em tempo de execução quando uma excepção chega ao topo da pilha de execução (ou seja, quando chega à rotina por onde o programa começou). Em linguagens concorrentes, parece-nos também claro que, normalmente, um processador deve terminar quando uma excepção é propagada até à sua rotina de criação.

Por outro lado, um programa concorrente tem geralmente vários processadores, cada um deles com um sub-programa associado. Parece-nos evidente que não seria aceitável que da falha de um processador resultasse a falha total do programa. Seria um pouco absurdo, fazendo uma analogia simples, que uma falha numa máquina de sumos de um aeroporto levasse a que este ficasse indisponível para qualquer outro uso (como por exemplo viajar para algum lado de avião).

Num contexto orientado por objectos não são os processadores que mandam: são sim os objectos. Assim um programa concorrente só deve terminar completamente se nenhum dos seus processadores for capaz de desempenhar a sua tarefa, ou se houver uma ordem superior para que todos terminem (este último caso, mais relacionado com programas em tempo-real, não será abordado neste trabalho).

Em resumo, as excepções ao longo do seu trajecto (ao serem propagadas de um lado para outro), podem ir deixando objectos indisponíveis (falha de invariante), eventualmente recuperáveis posteriormente, podendo mesmo terminar a execução de processadores.

5.19 Serviços de classe

Os serviços de classe (secção 3.16), especialmente os atributos, interferem directamente com mecanismos de concorrência. Se uma classe com esse tipo de serviços tiver

[25] Em EIFFEL será nos blocos de **rescue**.
[26] Em EIFFEL com a instrução **retry**.

instâncias executadas por diferentes processadores (sejam ou não concorrentes), então esses serviços são partilhados por todos esses processadores, necessitando de ser devidamente sincronizados com um esquema de sincronismo intra-classe (que inclua todas as instâncias da classe). As interferências deste mecanismo podem ainda ser maiores se os serviços de classe forem partilhados com classes descendentes.

Para lidar com esse problema a linguagem JAVA, para além de um monitor por objecto, tem também um monitor por classe. É da responsabilidade do programador o uso correcto desses esquemas de sincronismo.

Dada a complexidade das interferências que este mecanismo parece provocar, e também devido à linguagem EIFFEL não ter este tipo de serviços, optámos por não o incluir na linguagem protótipo.

5.20 Serviços de execução única

Os serviços de execução única (secção 3.17) podem ser adaptados a linguagens concorrentes. No entanto, é necessário, caso esses serviços sejam partilhados entre vários processadores[27], que o sistema de compilação da linguagem sincronize o acesso a esses serviços independentemente do esquema de sincronismo intra-objecto (já que esses serviços poderão ser partilhados por todos os objectos que sejam instâncias de uma classe).

Estes serviços poderão ser utilizados em programas concorrentes como outra forma para diferentes processadores terem acesso a referências de objectos concorrentes.

5.21 Atributos locais a processadores

Um mecanismo que pode ser útil em programas concorrentes é a possibilidade de declarar atributos locais a processadores[28]. A utilização deste tipo de atributos em objectos partilhados seria completamente segura, independentemente do esquema de sincronismo intra-objecto implementado.

A ideia base por detrás deste mecanismo é muito simples. Sabendo-se que na execução concorrente de objectos a interferência entre processadores se deve ao facto de eles actuarem sobre um estado partilhado do objecto pelos mesmos, porque não permitir quando for importante que os objectos possam ter estados específicos para cada processador?

Na biblioteca POSIX-THREADS [Butenhof 97] que acrescenta concorrência à linguagem procedimental C, existem os chamados dados locais a cada `thread`[29], assentes na mesma ideia base (embora não adaptada nem aplicada a linguagens orientadas a objectos).

Caso se permita a definição de atributos locais a processadores em linguagens concorrentes orientadas a objectos, os serviços de objectos partilhados que apenas modifiquem

[27] Ou seja, se o contexto de execução do serviço incluir todo o programa.

[28] Este mecanismo não foi ainda adoptado na linguagem protótipo desenvolvida, devido a não termos encontrado uma forma simples para sintacticamente o expressar.

[29] *thread local data*

este tipo especial de atributos serão, do ponto de vista da concorrência intra-objecto, equivalentes aos serviços que somente observam o estado do objecto.

A implementação de esquemas de *caching*[30] em objectos será uma das várias aplicações interessantes deste mecanismo.

5.22 Síntese das interferências entre mecanismos

As tabelas 5.2 e 5.3 sintetizam algumas das interferências negativas e sinérgicas tratadas neste capítulo.

[30]Por exemplo, para guardar temporariamente resultados de consultas computacionalmente pesadas.

–	**Descrição:**	**Refs.:**
Atributos públicos modificáveis – **TDA**	A existência deste tipo de atributos faz com que não seja apenas o objecto o responsável por garantir o seu invariante obrigando à propagação do sincronismo interno a todos os clientes que o podem modificar	(página 64)
Objectos activos – **TDA**	A escolha das mensagens (serviços) a serem aceites pelo objecto poderá nada ter a ver com o TDA do mesmo, podendo fazer-se em tempos instáveis do objecto	(página 71)
Comunicação assíncrona **Espera por necessidade** – **TDA** **Programação por contrato**	Sendo as pré-condições asserções impostas aos clientes de um serviço, em caso de incumprimento cabe a estes assumir essa responsabilidade. Se a verificação desta asserção for assíncrona, perde-se esta importante distribuição de responsabilidades	(página 73)
Atributos públicos modificáveis – **Sincronização condicional**	A existência de atributos públicos pode obrigar a propagar o código de sincronismo condicional a todos os clientes que os possam utilizar	(página 94)
Sincronização explícita – **Herança**	Designadas por anomalias de herança, estas interferências negativas derivam da impossibilidade – nessa opção de sincronização – de reutilizar o sincronismo herdado	(página 99)

Tabela 5.2: Algumas interferências inseguras entre mecanismos concorrentes.

+	**Descrição:**	**Refs.:**
Procedimento criação do objecto + **Criação de processadores**	Quando se justifique, a criação de certos objectos pode também criar novos processadores	(página 67)
Comunicação entre objectos + **Comunicação entre processadores**	A comunicação entre objectos pode ser reutilizada como um mecanismo de comunicação entre processadores, bastando para tal que cada objecto pertença a um processador	(página 71)
Separação comandos e consultas + **comunicação síncrona e assíncrona**	Um comando é tipicamente um envio unidireccional de uma mensagem para um objecto, logo com facilidade se lhe pode atribuir um comportamento assíncrono. Uma operação de consulta, por outro lado, é bidireccional, logo presta-se a um comportamento síncrono	(página 73)
Asserções concorrentes + **Sincronização condicional**	Para que continue a fazer sentido, uma asserção que dependa de outro processador que não o que a está a testar, tem de ter um comportamento de espera condicional	(página 97)
Selecção algorítmica por condições concorrentes + **sincronização inter-objecto**	Estas instruções (que inclui as instruções condicionais, repetitivas e as pré-condições) só fazem sentido se o estado dos objectos envolvidos na condição se tornar, a partir desse instante, apenas dependente desse processador	(página 98)
Procedimento criação do objecto + **Recuperar objectos para um estado estável**	Uma vez que os procedimentos de criação de objectos são os únicos que, por definição, não necessitam que o invariante se verifique no inicio da sua execução, eles podem ser reaproveitados em mecanismos de recuperação de objectos em estados instáveis	(página 104)
Serviços de execução única + **Partilha de referências a objectos concorrentes**	Os serviços de execução única podem ser reutilizados para serem um mecanismo de partilha de objectos concorrentes	(página 105)

Tabela 5.3: Algumas interferências sinérgicas entre mecanismos concorrentes.

Capítulo 6

A Linguagem MP-Eiffel

6.1 Introdução

No capítulo anterior analisaram-se crítica e detalhadamente várias aproximações para integração de mecanismos de programação concorrente em linguagens orientadas por objectos, tendo-se feito, com as devidas justificações, várias escolhas nessas aproximações. Neste capítulo vai-se apresentar uma linguagem de programação – denominada MP-Eiffel: *Multi-Programming Eiffel* – onde essas ideias estão a ser aplicadas e experimentadas. As suas principais características são as seguintes:

- segurança estática[1];
- processadores abstractos;
- sincronização abstracta de objectos concorrentes;
- sincronização automática de objectos concorrentes;
- mecanismos de comunicação entre processadores por mensagens e por partilha de memória;
- sistema de tipos estático com anotações de concorrência;
- mecanismo concorrente de excepções (como descrito na secção 5.18);
- linguagem de controlo de concorrência para eventual escolha de concretizações de processadores e de esquemas de sincronismo intra-objecto.

No projecto desta linguagem optou-se por incluir integralmente a linguagem Eiffel [Meyer 92]. Esta opção resultou não só da abordagem rigorosa e cuidada que essa linguagem faz à programação por objectos (sem dúvida, a preferida pelo autor), mas também do facto de ser praticamente a única linguagem com mecanismos apropriados de suporte à programação por contrato. A programação por contrato (secção 3.12), é uma ferramenta essencial com o objectivo de maximizar a correcção no software, mas também como implementação prática dos TDA (secção 3.9) de cada classe. É opinião

[1] Os eventuais problemas de segurança estática da linguagem Eiffel relacionados com a co-variância de tipos nos argumentos de rotinas redefinidas não é aqui abordada, já que saem fora do âmbito deste trabalho.

do autor que a programação por objectos será sempre uma metodologia incompleta se não considerar a programação por contrato.

Esta opção levantou ainda um outro desafio interessante: maximizar a utilidade de módulos existentes em EIFFEL sem que tal limitasse o potencial de concorrência de programas em MP-EIFFEL. Ou seja, pretendia-se que fosse possível utilizar directamente classes sequenciais (desenvolvidas em EIFFEL) para criar objectos concorrentes. Esse objectivo foi conseguido tendo para tal contribuído a ortogonalidade e sinergia na integração dos mecanismos concorrentes.

Do ponto de vista estritamente sintáctico, o MP-EIFFEL acrescenta apenas três palavras reservadas ao EIFFEL: **shared**, **remote** e **trigger**.

Tal como o SCOOP, o MP-EIFFEL faz uma aproximação axiomática na definição dos mecanismos de concorrência. Assim, o estatuto concorrente dos objectos resulta directamente da semântica associada a cada mecanismo, cabendo ao sistema de compilação a garantia de segurança na utilização desses mecanismos e a respectiva implementação. Esta aproximação diverge da seguida na linguagem JAVA, onde o programador é chamado a assumir – senão toda – pelo menos uma parte significativa da responsabilidade de garantir correcção na utilização dos mecanismos concorrentes. Um exemplo claro dessa situação é a utilização explícita da anotação de sincronismo **synchronized** nos métodos que requerem acesso exclusivo a objectos concorrentes, ou então em alternativa ter em consideração o complexo modelo de memória da linguagem [Lea 00, página 90].

Uma das primeiras dificuldades na concepção da linguagem foi a selecção de abstracções apropriadas para concretizar os dois modelos de comunicação entre processadores: envio de mensagens e partilha de objectos. A primeira tentativa nesse sentido foi, naturalmente, arranjar mecanismos ortogonais entre si para cada um dos modelos. Pelas razões apresentadas no capítulo anterior (secção 5.2.1) optou-se desde o início por fazer uso de anotações no sistema de tipos para identificar objectos concorrentes.

Assim, no caso do modelo de partilha de objectos decidimos reutilizar a anotação de tipo **shared** introduzida por Brinch Hansen para monitores [BH 73, secção 7.2].

Um objecto de um tipo **shared** será então um objecto concorrente cujo acesso, feito da mesma forma que o acesso a objectos sequenciais, faz uso do modelo de comunicação por partilha de objectos.

No caso do modelo por envio de mensagens a anotação utilizada no SCOOP – **separate** – seria uma hipótese. No entanto, na nossa opinião essa palavra não expressa bem a propriedade de comunicação que se pretende abstrair. Essa propriedade deveria aproximar-se mais do conceito de invocação remota de serviços subjacente a esta forma de comunicação (secção 5.6.2). A escolha recaiu assim sobre a anotação **remote**.

Nesta altura levantaram-se vários problemas. Em primeiro lugar, embora se pudesse utilizar a invocação normal de serviços também para este mecanismo de comunicação entre processadores, tal opção não nos parecia correcta já que a semântica de comunicação é muito diferente (pode ser assíncrona, secção 5.6.3). A este problema acrescia a conveniência, como se argumentou na secção 5.8.1, de se poder definir uma interface diferente para a recepção de mensagens com origem noutros processadores. Por fim, a comunicação por mensagens entre processadores obriga a que os objectos remotos estejam inequivocamente associados a um único processador (receptor), pelo que, ou se

adoptava uma aproximação como o SCOOP em que há uma separação total entre os objectos de cada processador – para a qual o uso de objectos partilhados pareceria um pouco forçada – ou se arranjava uma semântica alternativa para os objectos remotos.

Como se verá mais à frente neste capítulo, todos estes problemas foram resolvidos – na nossa opinião de uma forma bastante elegante – acrescentando um novo grupo de abstracções de linguagem (que não são anotações de tipo) denominado por *triggers* (nos dois sentidos que esta palavra pode ter: a de ser um mecanismo de gatilho ou a acção de despoletar). Dessa opção resultou ainda um efeito sinérgico muito interessante que foi fazer com que entidades remotas pudessem ser também utilizadas no modelo de comunicação por partilha de objectos, mas com a restrição de por seu intermédio só se poder utilizar serviços de consulta puros.

6.2 Comunicação por partilha de objectos

Em MP-Eiffel, uma invocação normal de um serviço aplicada a uma entidade concorrente (**shared** ou **remote**) constitui uma comunicação entre processadores por partilha de objectos (como é justificado na página 74).

6.2.1 Objectos partilhados

Os objectos partilhados são objectos concorrentes que podem – desde que, é claro, se respeite o respectivo TDA – ser livremente observados e modificados por todos os processadores que a eles tenham acesso. Este tipo de objectos não pertence a nenhum processador em particular (nem mesmo ao processador responsável pela sua criação). Em MP-Eiffel estes objectos só podem ser referenciados por entidades com a anotação de tipo **shared**. As regras de atribuição de valor a entidades com tipo garantem que nunca um objecto partilhado possa estar associado a um entidade com tipo que não seja também ela partilhada (secção 6.5).

Na figura 6.1 apresenta-se um exemplo de aplicação de objectos partilhados. Os objectos partilhados (no caso, em princípio, será apenas um) nesse exemplo servem para implementar classes para fazer o registo logístico de informação interna de programas. Assim, temos um objecto partilhado do tipo `LOG_REGISTER` onde toda essa informação pode ser centralmente registada.

Uma das características importantes desta linguagem, aliás facilmente constatável neste pequeno exemplo, é a possibilidade de construir classes sem anotações de concorrência (`LOG_REGISTER`) – ou seja, literalmente em Eiffel – sem que tal impeça futuros usos de instâncias concorrentes dessas classes (no caso, objectos partilhados). Desta forma as anotações de concorrência podem ser restringidas apenas aonde são estritamente necessárias, potenciando as possibilidades de reutilização de classes, e facilitando a compreensão dos programas. Repare-se ainda que basta uma única anotação de tipo aplicada a uma classe normal para termos um objecto concorrente (sem o peso das redundâncias existentes, por exemplo, no SCOOP).

```
class LOG_REGISTER

inherit
  LOG_USER;

creation
  make;

feature

  make(filename: STRING) is
    require
      not is_logging
    do
      ... -- open file handler
    end;

  start is
    do
      log(Current,"Starting logging...");
      is_logging := true
    end;

  stop is
    do
      log(Current,"Stopping logging...");
      is_logging := false
    end;

  is_logging: BOOLEAN;

feature

  log(source: LOG_USER;message: STRING)
    require
      source /= Void;
      message /= Void;
      is_logging
    do
      file.writeln_array_string(
        <<"[",current_date.to_string,
          "] ",source.id,": ",message>>);
    end;

  ...

end -- LOG_REGISTER
```

```
deferred class LOG_USER

feature

  id: STRING is
    do
      Result := class_name
    end;

end -- LOG_USER
```

```
class EXAMPLE_LOG

inherit
  LOG_USER

feature

  set_log_register(log_reg: shared LOG_REGISTER) is
    require
      log_reg /= Void
    do
      log_register := log_reg
    end;

  log_register: shared LOG_REGISTER;

  foo is
    do
      log_register.log(Current,"Hello world!");
    end;

end -- EXAMPLE_LOG
```

Figura 6.1: Exemplo de utilização de objectos partilhados.

6.2.2 Objectos remotos

Tal como os objectos partilhados, os objectos remotos são também objectos concorrentes, mas diferem em dois aspectos essenciais: pertencem a um (único) processador, e apenas podem ser observados (sem efeitos colaterais) pelos restantes processadores que a ele tenham acesso. As regras do sistema de tipos da linguagem permitem que esses objectos possam ser referenciados por outros processadores, para além do seu criador, mas impedem estaticamente qualquer tentativa de modificação desses objectos por processadores remotos (ou seja, só permitem a invocação de consultas puras).

Na figura 6.2 apresenta-se um exemplo de aplicação destes objectos[2]. No problema em questão existe uma classe – `EARTH` – onde o estado de algumas variáveis climáticas (no exemplo, o valor da temperatura e o vector velocidade do vento) pode ser acedido em tempo real (o tempo é implícito neste exemplo). Por outro lado, existe também uma classe que abstrai uma estação atmosférica, que periodicamente recolhe essas informações da classe `EARTH`. Uma vez que a estação atmosférica não afecta, nem pode afectar, o comportamento da classe `EARTH`, e sendo que podem existir várias estações a recolher informação, faz todo o sentido que esses objectos tenham uma referência remota da instância da classe `EARTH`.

É importante referir-se que a sincronização intra-objecto de objectos remotos é bastante menos exigente que a de objectos partilhados. Um esquema de sincronismo leitores-escritor concorrentes (secção 5.10.5) dá total disponibilidade concorrente a estes objectos.

6.2.3 Sincronização

Esta linguagem adopta integralmente os mecanismos e soluções descritas no capítulo anterior para os vários aspectos de sincronismo:

Sincronização intra-objecto: é abstracta (secção 5.9.1) e automática (secção 5.10), podendo o programador participar na escolha do esquema de sincronismo através de uma linguagem de controlo de concorrência (secção 6.7).

Sincronização inter-objecto: é feita quando há selecção algorítmica por condições concorrentes, como é descrito na secção 5.15.

Sincronização condicional: é feita por asserções concorrentes (secção 5.14).

6.3 Comunicação por envio de mensagens: *Triggers*

A linguagem MP-Eiffel implementa a comunicação por envio de mensagens entre processadores através de um conjunto de mecanismos denominados por *triggers*, em que a identificação dos processadores é indirecta (secção 5.6.2).

Um *trigger* é uma mensagem directa entre processadores. Para que essa comunicação se possa fazer é necessário ter processadores capazes de receber essas mensagens

[2] O mesmo problema pode ser melhor resolvido com o recurso não só a objectos remotos mas também a *triggers*, como se verá mais à frente.

```
class ATMOSPHERIC_STATION

feature

  valid_longitude(long: REAL): BOOLEAN is
    do
      Result := long >= -180.0 and long <= 180.0
    end;

  valid_latitude(lat: REAL): BOOLEAN is
    do
      Result := lat >= -90.0 and lat <= 90.0
    end;

  valid_altitude(alt: REAL): BOOLEAN is
    do
      Result := alt >= 0.0
    end;

  longitude,latitude,altitude: REAL;

  set_position(long,lat,alt: REAL) is
    require
      valid_longitude(long);
      valid_latitude(lat);
      valid_altitude(alt);
    do
      longitude := long;
      latitude := lat;
      altitude := alt;
      position_defined := true
    end;

  position_defined: BOOLEAN;

  earth: remote EARTH;

  set_earth(the_earth: remote EARTH) is
    require
      the_earth /= Void
    do
      earth := the_earth
    end;

  earth_defined: BOOLEAN is
    do
      Result := earth /= Void
    end;

  start(sampling_period,num_iters: INTEGER) is
    require
      position_defined;
      earth_defined
    do
      from i := 1 until i > num_iters loop
        fetch_data;
        wait(sampling_period);
        i := i + 1
      end
    end;

end -- ATMOSPHERIC_STATION
```

```
class EARTH

feature

  temperature(long,lat,alt: REAL): REAL is
      -- real-time temperature value
    require
      valid_longitude(long);
      valid_latitude(lat);
      valid_altitude(alt);
    do
      ...
    end;

  wind_speed(long,lat,alt: REAL): VECTOR[REAL] is
    require
      valid_longitude(long);
      valid_latitude(lat);
      valid_altitude(alt);
    do
      ...
    end;

end -- EARTH
```

Figura 6.2: Exemplo de utilização de objectos remotos.

```
class C

trigger
  tick

feature

  tick is
    do
      ...
    end

end -- C
```

Figura 6.3: Exemplo de declaração de *triggers*.

(*triggers*, no sentido de gatilhos), e uma instrução apropriada para as enviar (*trigger*, no sentido de despoletar).

Para que um processador possa receber mensagens de outros processadores terá de ter a si associados objectos cujas classes explicitamente declarem alguns dos seus serviços como *triggers*. Essa associação é feita simplesmente pela criação desses objectos (que não poderão ser de um tipo **shared** nem **remote**) pelo processador (esses objectos passarão a pertencer ao processador).

Uma declaração de *trigger* é sintacticamente idêntica à declaração de construtores em EIFFEL, com a diferença de a palavra reservada utilizada ser **trigger** (e não **creation**). A figura 6.3 exemplifica uma declaração de *triggers*. Objectos da classe `C` (ou descendentes) podem receber invocações remotas ao seu serviço `tick`. Sendo que a execução desses serviços cabe ao processador que criou o objecto.

O envio de *triggers* é feito através da instrução de invocação de *triggers*, que difere sintacticamente de uma invocação normal de serviços de um objecto somente por essa invocação ser precedida da palavra reservada **trigger**.

```
x: remote C;
...
trigger x.tick;
```

Uma vez que este modelo de comunicação obriga à identificação inequívoca do processador receptor, os *triggers* só farão sentido se forem enviados para objectos remotos (ou seja objectos associados a entidades com tipo remotas). Estes são os únicos objectos que podem pertencer a outros processadores que não o que envia a mensagem e aos quais está associado um processador. Assim, a entidade `x` no exemplo dado terá obrigatoriamente de ser remota.

Ao contrário da declaração de serviços de criação, os triggers são herdados em classes descendentes, podendo mesmo os seus nomes serem alterados com o mecanismo de mudança de nomes da linguagem EIFFEL [Meyer 92, página 81]. Assim, as relações

de subtipo são perfeitamente compatíveis com *triggers*, não havendo lugar (para além do problema da covariância) a interferências inseguras entre ambos os mecanismos.

A linguagem garante que nenhum *trigger* é perdido, e que eles são normalmente atendidos por ordem de chegada. Essa ordem pode, no entanto, ser alterada (pelo sistema de escalonamento de mensagens do processador receptor) desde que se mantenha a consistência sequencial das mensagens (página 65). Futuramente poderá haver a possibilidade de definir prioridades diferentes para diferentes *triggers* através da linguagem de controlo da concorrência, mas são adaptações do mecanismo que ainda não estão devidamente pensadas.

Como é óbvio, os processadores receptores de *triggers* só podem executar uma dessas mensagens de cada vez. Essa execução só pode ter lugar quando o processador estiver disponível para a execução, ou seja quando ele estiver num estado de espera (a secção 6.4 descreve as diferentes fases do tempo de vida dos processadores).

A figura 6.4 mostra uma implementação com *triggers* para o problema de estações atmosféricas apresentado atrás (página 113). Podemos verificar que a introdução de *triggers* permitiu separar facilmente o problema de activação periódica das estações atmosféricas (feita com o recurso a um metrónomo), da observação sobre o estado do objecto `earth`. Dessa forma, a qualidade da solução é melhorada e com facilidade se pode acrescentar novas funcionalidades (como seja o serviço de paragem: `stop`).

6.3.1 *Triggers* síncronos e assíncronos

A execução dos *triggers* tanto pode ser síncrona como assíncrona dependendo do serviço remotamente requerido. Assim, como é explicado na página 73, a execução de serviços de consulta será síncrona, e a execução de comandos será assíncrona.

Uma consequência interessante desta diferença de comportamento, resulta da possibilidade, permitida em MP-Eiffel, de enviar *triggers* para o próprio processador. No caso de um serviço de consulta, o resultado da instrução de *trigger* não se vai diferenciar da invocação directa do serviço. Já no caso do *trigger* de comandos, o processamento só terá lugar quando o *trigger* for escalonado para execução na fase de espera do processador (o que pode ser útil em algumas situações).

6.3.2 *Triggers* e encapsulamento de informação

A interface dos *triggers* é dada – não pela cláusula **feature** onde o serviço que lhe está associado é declarado e eventualmente implementado – mas sim directamente na cláusula de declaração dos *triggers*[3]. Assim, um serviço que esteja associado a um *trigger* tem duas interfaces distintas: uma para clientes normais do respectivo objecto e outra para *triggers* (ver figura 6.5).

6.3.3 Argumentos formais de *triggers*

Nada impede que os *triggers* estejam ligados a rotinas da classe que tenham argumentos formais. No entanto, como se pode facilmente constatar, os tipos desses

[3]De forma perfeitamente similar e consistente ao que acontece com os construtores do objecto.

```
-- MP-Eiffel library class

remote class METRONOME

creation
  begin_ticking;

feature

  begin_ticking is -- new processor
    do ... end; -- ticks registered
                -- METRONOME_RECEIVER's

  stop_ticking is -- ends processor
    do ... end;

feature

  user_exists(user: remote METRONOME_RECEIVER):
      BOOLEAN is
    do ... end;

  start(user: remote METRONOME_RECEIVER;
        period: INTEGER) is
    do ... end;

  stop(user: remote METRONOME_RECEIVER) is
    require
      user_exists(user)
    do ... end;

end -- METRONOME
```

```
-- MP-Eiffel library class

deferred class METRONOME_RECEIVER

trigger
  tick

feature

  tick is
    deferred
    end;

end -- METRONOME_RECEIVER
```

```
class ATMOSPHERIC_STATION

inherit
  METRONOME_RECEIVER;

feature

  (...)
  -- valid_longitude, valid_latitude, valid_altitude
  -- longitude,latitude,altitude
  -- set_position, position_defined
  -- earth, set_earth, earth_defined

  metronome: remote METRONOME;

  set_metronome(the_metronome: remote METRONOME) is
    require
      the_metronome /= Void
    do
      metronome := the_metronome
    end;

  metronome_defined: BOOLEAN is
    do
      Result := metronome /= Void
    end;

  working: BOOLEAN;

  start(sampling_period: INTEGER) is
    require
      not working;
      position_defined;
      metronome_defined;
      earth_defined
    do
      working := true;
      trigger metronome.start(Current,
                              sampling_period)
    end;

  stop is
    require
      working;
    do
      working := false;
      trigger metronome.stop(Current)
    end;

feature {METRONOME}

  tick is
    do
      file.writeln_array_string(
        <<"Temperature at ",current_date.to_string,
          " is ",earth.temperature.to_string,".">>);
    end

end -- ATMOSPHERIC_STATION
```

Figura 6.4: Exemplo de utilização de *triggers*.

```
class C

trigger {X} -- only X descendants can trigger foo
  foo;

trigger     -- anyone can trigger bar
  bar;

feature {Y} -- only Y descendants can call foo

  foo is
    do
       ...
    end;

end -- C
```

Figura 6.5: Exemplo de declaração de *triggers* com encapsulamento.

argumentos formais terão de ser sujeitos a algumas restrições: ou são de um tipo expandido, ou de um tipo concorrente (**shared** ou **remote**). Não faria sentido ter um argumento formal não expandido e não concorrente num *trigger*, já que tal implicaria que o processador remoto que quisesse invocar esse *trigger* teria de passar como argumento uma referência para um objecto local ao próprio processador receptor (o que é uma impossibilidade). Se o argumento for expandido[4] [Meyer 92, página 194], aplica-se uma semântica de cópia integral do objecto (passagem por valor), pelo que o problema não se coloca.

6.4 Processadores

Em MP-Eiffel os processadores são criados implicitamente sempre que a instrução de criação de objectos é aplicada sobre uma entidade remota. Esta opção é consistente com a semântica dos objectos remotos já que estes, por definição, pertencem a outro processador. Logo, a criação de um objecto remoto, por ser um comando, terá de implicar a criação prévia do processador que o vai executar.

Normalmente um processador existe desde que é criado até que o respectivo subprograma termina (serviço de criação seleccionado). Isso não acontece, no entanto, caso os processadores tenham a si associados *triggers*. Nessa situação, esses processadores ficarão num estado de espera podendo ser acordados pela invocação remota de um dos seus *triggers*, ou terminados quando o programa termina. A figura 6.6 mostra o diagrama de estados completo do tempo de vida de um processador.

[4]Ou para ser mais rigoroso: completamente expandido.

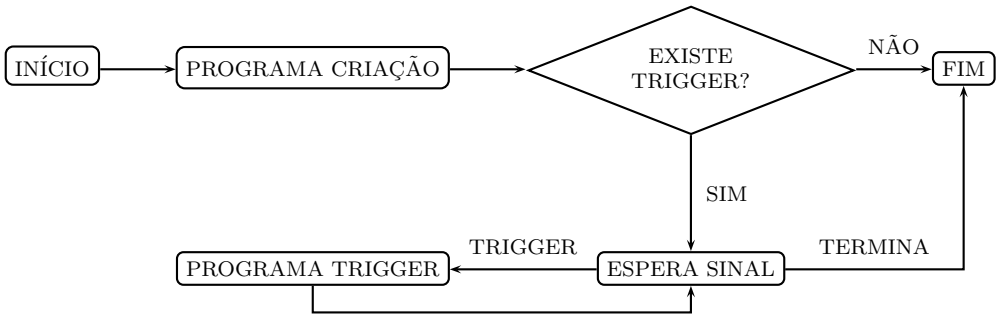

Figura 6.6: Vida de um processador.

6.5 Sistema de tipos

O sistema de tipos do MP-Eiffel é seguro quanto às anotações de concorrência[5]. As regras que garantem essa segurança estática são as seguintes. Seja `x` uma entidade com tipo à qual se possa atribuir um valor (um atributo, um argumento formal ou uma variável local), e `expr` uma expressão qualquer, tal que o tipo de `expr` é conforme com o tipo de `x`. Em MP-Eiffel a expressão `expr` pode ser atribuída a `x`,

```
x := expr,
```

numa das seguintes condições:

1. se `x` e `expr` forem ambos partilhados; ou ambos remotos; ou ambos sem anotações de concorrência;

2. se `x` for remoto e `expr` não tiver nenhuma anotação de concorrência (os objectos que possam estar associados a `expr` passam a ser concorrentes).

3. se `x` for expandido (desde que não contenha, directa ou indirectamente, nenhum atributo que seja uma referência).

Como se pode constatar comparando com as restrições de subtipo referidas no capítulo anterior, secção 5.17.2, existe aqui uma aparente contradição com a regra 2. Nessa secção é referido (e bem) que é inseguro atribuir a uma entidade concorrente um objecto sequencial, ou seja, exactamente o que a regra 2 parece propor. O problema é resolvido em MP-Eiffel pelo sistema de compilação. De facto o objecto associado a `expr` não pode ser sequencial (tem de estar sincronizado), cabendo ao sistema de compilação a detecção de todos esses objectos.

Infelizmente, este comportamento desejável ainda não foi implementado no compilador de MP-Eiffel devido à complexidade do sistema de tipos da linguagem Eiffel[6], estando a ser feita uma implementação com uma outra anotação de tipo – **visible** – feita especificamente para este propósito. Assim, neste momento, a regra 2 é:

2. se `x` for remoto e `expr` visível.

[5]Persistem ainda alguns buracos herdados da linguagem Eiffel, como referido na página 24.

[6]Em particular, a existência de "âncoras" [Meyer 92, página 211] torna a verificação de tipos pelo compilador um pouco mais complicada.

```
r_proc is
  once {processor}
    ...
  end

r_all is
  once
    ...
  end
```

```
r_proc_obj is
  once {processor,object}
    ...
  end

r_proc_all is
  once {object}
    ...
  end
```

Figura 6.7: Exemplo de serviços de execução única.

6.6 Serviços de execução única

A linguagem MP-Eiffel permite definir cinco diferentes contextos de execução para os serviços de execução única: programa ou processador, objecto ou classe, e chave livre. É possível também fazer combinações entre estes contextos de execução, excepto programa–processador e objecto–classe (seria uma contradição de termos). Por omissão, o contexto de execução destes serviços é por processador e por classe.

Como foi referido na secção 5.20, este tipo de serviços, caso incluam o programa como contexto de execução (ou seja, uma partilha entre todos os processadores), requerem uma sincronização apropriada. É necessário também prevenir interferências inseguras com outros mecanismos da linguagem. No caso do MP-Eiffel pode haver interferências inseguras com o tipo das entidades eventualmente utilizadas na invocação destes serviços (argumentos formais e o resultado das funções).

A regra de segurança (estática) é simples. No caso de serviços de execução única em que o contexto de execução inclui todo o programa, só é permitido que os tipos dos argumentos formais e, no caso das funções, que os tipos dos respectivos resultados sejam (completamente) expandidos ou concorrentes (partilhado ou remoto). Desta forma garante-se que uma referência não concorrente possa ser visível por vários processadores.

Contextos de execução que não incluam todo o programa não interferem minimamente com os mecanismos de concorrência (funcionam exactamente como em linguagens sequenciais).

Estes serviços são também muito úteis em programas concorrentes já que dão mais uma possibilidade elegante de dar acesso a objectos partilhados e remotos. A figura 6.7 exemplifica a declaração de alguns destes serviços.

6.7 Linguagem de controlo de concorrência

Uma das características marcantes da linguagem MP-Eiffel é o facto de relegar para fora do seus programas aspectos que tenham a ver com implementações em particular de mecanismos de concorrência como sejam os esquemas de sincronismo de objectos concorrentes ou a atribuição de prioridades diferentes no acesso a recursos partilhados.

```
synchronize class X
  default: crw; -- concurrent readers-writer
  procedure_one,procedure_two,
  procedure_three: lockfree
end
```

```
synchronize local entity a
  at some_method in class X;
  default: exrw
end
```

Figura 6.8: Exemplo sincronismo utilizando MP-Eiffel-CCL.

Para esse efeito está ser pensada uma linguagem de suporte ao sistema de compilação onde esses aspectos podem ser definidos e adaptados a diferentes contextos de execução. Essa linguagem é designada por Linguagem de Controlo de Concorrência do MP-Eiffel (*Concurrency Control Language*): MP-Eiffel-CCL.

Nessa linguagem, para a escolha do sincronismo intra-objecto, existem quatro anotações reservadas para cada um dos esquemas de sincronismo possíveis: **monitor**, **exrw** (exclusão leitores-escritor), **crw** (leitores-escritor concorrentes), **lockfree** (livre de bloqueamento). Poderá fazer-se a especificação desses esquemas de sincronismo, ou às classes como um todo, ou somente às entidades através das quais os objectos são criados. A figura 6.8 exemplifica essas duas situações.

A especificação de esquemas mistos de sincronismo faz-se declarando os serviços do objecto aos quais se quer associar tipos específicos de sincronismo. O sistema de compilação da linguagem MP-Eiffel encarrega-se de verificar a validade e a exequibilidade da especificação proposta.

Capítulo 7

Conclusões

Neste trabalho fez-se uma aproximação sistemática à construção de linguagens orientadas por objectos concorrentes. Para tornar claras e objectivas as várias escolhas que foram sendo feitas nesse processo, houve o cuidado de definir critérios de qualidade de linguagens (capítulo 2). Desde o início que se pretendeu integrar a programação concorrente em linguagens orientadas por objectos (e não o inverso), pelo que, no capítulo 3 se apresentou detalhadamente esse tipo de programação. Reconhecendo a existência de muitas variantes para este tipo de linguagens, ao ponto de poderem ter diferenças importantes nos métodos de programação que se lhes aplicam, houve o cuidado, nesse mesmo capítulo, de não só identificar algumas dessas diferenças, como também de tornar claras as escolhas de base feitas para este trabalho. No capítulo 4 apresentaram-se os requisitos colocados pela programação concorrente. O capítulo 5, onde se concentrou a maioria das contribuições feitas, faz uma aproximação sistemática, e em grande medida objectiva, à integração de mecanismos de concorrência em linguagens orientadas por objectos (dando sempre prioridade à metodologia de programação orientada por objectos). Por fim no capítulo 6 apresentou-se uma linguagem protótipo – MP-Eiffel – onde todas as escolhas e funcionalidade descritas no capítulo 5 foram integradas. Relativamente ao resultado final representado por essa linguagem, há a destacar a segurança estática, a expressividade e a abstracção dos mecanismos propostos assim como o elevado grau de integração sinérgica conseguida em muitos casos.

7.1 Contribuições

Neste trabalho foram feitas as seguintes contribuições:

- Aproximação sistemática e objectiva à integração de mecanismos concorrentes em linguagens orientadas por objectos (capítulo 5);
- Sincronização abstracta de objectos concorrentes (secção 5.9.1);
- Sincronização automática de objectos concorrentes (secções 5.10, 5.11 e 5.12);
- Esquemas mistos de sincronismo intra-objecto automático (secção 5.10.7);

- Solução para a integração automática do sincronismo intra-objecto e o sincronismo inter-objecto em objectos concorrentes (secção 5.12.3);
- Comportamento seguro de asserções concorrentes[1] (secção 5.14);
- Proposta para expressar de uma forma sinérgica e segura o sincronismo inter-objecto (secção 5.15);
- Mecanismo disciplinado de excepções concorrentes (secção 5.18);
- Integração sinérgica, na linguagem MP-Eiffel, de abstracções para ambos os modelos de comunicação entre processadores (secções 6.2 e 6.3).

7.2 Trabalho futuro

No fim de um trabalho como este temos a sensação de que muito ainda haveria a fazer apesar de tudo o que foi realizado. Sem dúvida que o aspecto mais frustrante (para quem assume ter gosto em ser engenheiro) foi a incapacidade do autor em conseguir ter um sistema de compilação completo e utilizável de uma forma segura para a linguagem protótipo proposta no capítulo 6. A finalização desse sistema (que se espera não ser demorada) será a principal prioridade para o trabalho a realizar no futuro, até porque o autor está convencido que as características que se julga interessantes e poderosas da linguagem a tornarão alvo de interesse (mais não seja para contribuir para o aparecimento de outras linguagens concorrentes mais expressivas e seguras).

A integração de requisitos de tempo-real em linguagens concorrentes será uma segunda área que se espera vir a desenvolver. O facto de existirem muito poucas aproximações linguísticas a esta área da programação, e também a existência de um grupo de investigação neste domínio em forte crescimento no departamento a que pertenço, tornam esse desafio mais interessante e com boas perspectivas de poder vir a ser bem sucedido. Por outro lado, as características da aproximação proposta parecem ser uma base apropriada para a integração de mecanismos de tempo-real (veremos se assim o será).

Por fim, pretendemos definir com mais rigor a linguagem de controlo de concorrência e fazer a sua implementação. Dessa forma cumpre-se um dos objectivos propostos com esta aproximação (partilhado com a aproximação SCOOP) – os processadores abstractos – e poderá também facilitar-se a integração de mecanismos de tempo-real.

[1] Muito embora, na opinião do autor, se trate apenas de uma generalização da proposta de Meyer para as pré-condições concorrentes.

Apêndice A

Introdução à linguagem SCOOP

A linguagem SCOOP [Meyer 97, capítulo 30] é uma proposta para extender a linguagem EIFFEL com mecanismos de concorrência.

A.1 Abordagem explícita à concorrência

O sistema de tipos é utilizado para a abordagem explícita de concorrência feita no SCOOP. Para esse efeito foi acrescentada uma anotação de tipo através da palavra reservada **separate**. Aliás, esta é a única palavra reservada acrescentada à linguagem EIFFEL, sendo que essa pequena diferença sintáctica é suficiente para o aparecimento de um conjunto bastante rico de mecanismos concorrentes.

A.2 Criação de processadores

Para criar um novo processador basta utilizar a instrução de criação de um novo objecto sobre uma entidade declarada como separada. Esse novo processador irá executar o serviço de criação (caso algum seja seleccionado), ficando depois disponível para executar serviços do objecto como resposta a invocações de outros processadores.

A.3 Comunicação entre processadores

A comunicação entre processadores segue exclusivamente o modelo de comunicação por envio de mensagens. Essa comunicação faz-se de forma similar à invocação qualificada de serviços de objectos, com a diferença de que a invocação se aplica a uma entidade separada.

```
x.f(y)
```

Assim, se o tipo da entidade `x` for **separate** e se a essa entidade estiver ligado um objecto separado (o SCOOP permite que a uma entidade separada esteja ligado um objecto não separado [Meyer 97, página 973]), o processador ao qual pertence o objecto actual (`Current`) estará a enviar uma mensagem para o processor do objecto ligado a essa entidade.

A regra de invocações separadas [Meyer 97, página 985] obriga a que só argumentos formais separados possam ser utilizados como destino de invocações separadas.

A.4 Processadores abstractos

Os processadores não estão ligados a um suporte de execução específico. Assim, recorrendo a um ficheiro de controlo de concorrência [Meyer 97, página 971] é possível associar cada processador existente em programas a um suporte de execução que esteja disponível.

A.5 Sincronismo intra-objecto

Em SCOOP qualquer objecto pertence a um único processador (embora um processador possa ter muitos objectos), sendo apenas permitida a execução de serviços do objecto nesse processor. Assim, na sua proposta original, não é permitida a existência de concorrência intra-objecto, havendo disponibilidade do objecto apenas para o processador que o criou.

A.6 Sincronismo inter-objecto

Na sua proposta original (semântica de invocações separadas [Meyer 97, página 996]), todos os objectos ligados a argumentos formais separados são reservados exclusivamente durante toda a execução da rotina. Esse execução será, se necessário, adiada (bloqueada) até que tal exigência seja cumprida.

Uma proposta mais recente [Meyer 05, Nienaltowski 06a] impõe essa semântica apenas a argumentos formais que sejam ligados[1] (a definição de entidades ligadas pode ser encontrada aqui [ECMA-367 05, página 75]). No caso de os argumentos formais não serem ligados então não há lugar à reserva de eventuais objectos que possam ser referenciados por esses argumentos.

A.7 Sincronismo condicional

O sincronismo condicional de objectos é feito recorrendo a pré-condições aplicadas a argumentos formais separados. Nessa situação a reserva dos objectos depende não só da sua disponibilidade como também da verificação da pré-condição separada. As pré-condições que envolvem argumentos formais separados são assim designadas por pré-condições concorrentes e o seu comportamento é similar a uma espera condicional.

[1] *attached*.

Apêndice B

Considerações Sobre a Implementação da Linguagem MP-Eiffel

Este apêndice aborda as soluções encontradas para implementar algumas das funcionalidades do sistema de compilação. Pelo facto de a linguagem MP-Eiffel ter uma aproximação axiomática à concorrência, especialmente na automatização do sincronismo dos objectos concorrentes, a realização do sistema de compilação levantou alguns problemas que se julga suficientemente interessantes para aqui serem apresentados.

De qualquer forma chama-se a atenção de que as soluções aqui apresentadas (e implementadas) são apenas umas de várias possíveis aproximações práticas, que servem essencialmente para demonstrar a realizabilidade dos mecanismos propostos, e para testar o protótipo da linguagem. Muito trabalho falta ainda realizar, por forma a que o sistema de compilação se possa considerar utilizável para programar em MP-Eiffel.

B.1 Enquadramento

Uma vez que este trabalho se insere no estudo de mecanismos concorrentes para linguagens orientadas por objectos, os quais requerem ainda uma adequada experimentação prática, optou-se por facilitar tanto quanto possível a implementação do sistema de compilação "minimizando" o tempo da sua implementação, em detrimento do tempo de compilação e também – em certos casos – do tempo de execução dos programas em MP-Eiffel.

Uma das opções tomadas de início consistiu em se restringir a plataforma de execução do sistema de compilação da linguagem a um único sistema operativo – o Linux – e a um único suporte de execução concorrente de processadores – a biblioteca POSIX-Threads para a linguagem C.

Outra das opções iniciais assentou na simplificação do sistema de compilação escolhendo-se como linguagem objectivo a que está mais próxima do MP-Eiffel. Assim o sistema de compilação gera código em Eiffel. Esse código é depois compilado utilizando um compilador Eiffel do domínio público (SmallEiffel).

B.1.1 *Thread-Safe SmallEiffel*

O compilador SmallEiffel nasceu em 1995 como projecto de implementação de uma versão de código-aberto de um compilador de Eiffel[1]. Em 2002, o SmallEiffel é "abandonado" pelos seus implementadores em benefício de uma nova linha de desenvolvimento do compilador, denominada então por SmartEiffel (na qual era pretendida a implementação dos mecanismos de concorrência propostos por Meyer no modelo SCOOP).

No âmbito do trabalho desta tese, e uma vez que o código (C) gerado pelo compilador SmallEiffel não era seguro para ser compilado e executado com a biblioteca POSIX-Threads existente no Linux, o autor desta tese em 2000, alterou o compilador por forma a que o código gerado fosse seguro. Da mesma forma, criou-se uma biblioteca em Eiffel de encapsulamento da biblioteca POSIX-Threads. Esse "novo" compilador foi designado por Thread-Safe SmallEiffel[2]. O apêndice D contém uma descrição dessa biblioteca desenvolvida sobre o SmallEiffel.

Após o aparecimento do SmartEiffel, e uma vez que este pretende implementar o modelo SCOOP, optou-se por não adaptar a versão segura para essa nova linha de desenvolvimento do compilador de Eiffel.

B.1.2 *PCCTS*

A construção do compilador de MP-Eiffel assentou num grupo de ferramentas para gerar analisadores léxicos e sintácticos designado por PCCTS[3].

B.2 Detecção de objectos concorrentes

Um dos problemas mais difíceis levantados na implementação do sistema de compilação do MP-Eiffel consiste na localização em tempo de compilação – sem falhas de segurança nem excesso de falsos positivos[4] – dos objectos concorrentes. O sistema de tipos do MP-Eiffel foi pensado, desde o início, de forma a não só tornar este problema possível mas também tratável. Só assim se torna possível a implementação automática e segura do sincronismo de objectos concorrentes por parte do sistema de compilação, sem penalizar a implementação dos restantes objectos sequenciais (que, num programa normal, tenderão a ser a larguíssima maioria).

Como já foi referido no capítulo 3, os programas não manipulam directamente os objectos. Estes são criados e utilizados através de entidades com tipo desse programa, ou seja, através de: atributos, funções, variáveis locais e argumentos formais de procedimentos ou funções. Assim, em MP-Eiffel um objecto será concorrente se, e só se, ele puder estar associado a uma entidade concorrente.

Para que uma entidade possa ser concorrente é condição necessária que a mesma seja uma referência, ou contenha directa ou indirectamente um atributo que seja ele próprio uma referência. Se uma entidade for completamente expandida (como acontece

[1] Na altura não existia nenhum outro compilador livre.

[2] É do domínio público e está disponível em `http://www.ieeta.pt/~mos/thread-safe-se/index.html`

[3] *Pardue Compiler Construction Tool Set.*

[4] Ou seja, sem anotar objectos puramente sequenciais como sendo concorrentes.

com alguns dos tipos básicos do EIFFEL como o `INTEGER`, `REAL` e o `BOOLEAN`), então – como a semântica de atribuição de valor a essas entidades implica sempre à cópia integral do objecto – esse novo objecto não será concorrente.

Para além desta condição, uma entidade só será concorrente se uma das situações seguintes se verificar:

1. se a entidade for partilhada (tipo **shared**);
2. se a entidade for remota (tipo **remote**);
3. se for uma entidade normal, e for passada como parâmetro de um argumento formal remoto do procedimento de criação de um novo processador;
4. se for uma entidade normal, e for passada como parâmetro de um argumento formal remoto na invocação de um *trigger* de um outro processador;
5. se for uma entidade normal e se for acedida, directa ou indirectamente, passando por uma entidade partilhada ou remota.

As duas primeiras situações são, por definição, evidentes: entidades do tipo partilhado ou remoto são concorrentes. As restantes situações, são um pouco mais complicadas e estão directamente relacionadas com a semântica dos objectos remotos. Quando uma entidade remota no programa de um processador está associada a um objecto, esse objecto será (necessariamente) um objecto normal de outro processador, e como tal, provavelmente estará associado a entidades normais do programa desse outro processador. Por este facto, essas entidades normais existentes no programa do processador dono desse objecto (concorrente), terão também de ser entidades concorrentes (embora, com a propriedade muito importante de apenas serem modificáveis por um único processador, pelo que o seu comportamento externo é semanticamente equivalente a objectos sequenciais).

Como por definição, uma entidade remota só pode invocar serviços sem efeitos colaterais dos objectos aos quais esteja associada, essas entidades não podem ser directamente definidas pelos programas de processadores que não o processador ao qual pertencem, ou seja, em cujo programa estão declaradas. Por exemplo, o código apresentado na figura B.1 – embora à primeira vista possa parecer correcto – não é um programa válido:

O erro neste programa reside na invocação de um procedimento através de uma entidade remota.

Para que seja o programa associado ao respectivo processador o responsável pela associação das entidades remotas, só existem três possibilidades:

- aquando da criação desse novo processador;
- utilizando *triggers*;
- através de outro objecto remoto.

Todas estas possibilidades são consentâneas com a semântica esperada das entidades remotas. Passar a referência de um objecto normal aquando da criação de um novo

```
-- assume this class to be part of
-- processor 1's program
class A_PROC1_CLASS

  -- ...

  proc2: remote A_PROC2_CLASS;

  abc is
    local
      obj: CLASS_X -- normal entity
    do
      create obj;
      proc2.def(obj); -- incorrect call!
    end

end -- A_PROC1_CLASS
```

```
-- assume this class to be part of
-- processor 2's program
class A_PROC2_CLASS

  -- ...

  remote_obj: remote CLASS_X;

  def(rem_obj: remote CLASS_X) is
    do
      remote_obj := rem_obj;
      -- ...
    end

end -- A_PROC2_CLASS
```

Figura B.1: Programa errado.

processador, consiste na definição do seu estado de execução inicial. Invocar um *trigger*, é formalmente equivalente a uma invocação remota de um serviço, pelo que nada impede que esse serviço tenha efeitos colaterais para o respectivo processador remoto. Por fim, o uso de uma referência pré-existente de um objecto remoto, para se aceder a referências de outros objectos remotos, não tem efeitos colaterais no processador dono desse objecto remoto, pelo que é um uso normal de um serviço do objecto.

A detecção de objectos concorrentes, e uma vez que apenas existem essas três possibilidades para associar entidades remotas a objectos, resolve-se propagando a propriedade concorrente a todas as entidades normais que sejam utilizadas nessas três situações.

O programa seguinte exemplifica a utilização do procedimento de criação de um novo processador para passagem da referência de um objecto remoto.

```
class A_PROCESSOR

creation
  make

feature{NONE}

  make(obj: remote CLASS_X) is
    do
      ...
    end;

end -- A_PROCESSOR
```

```
class SOMEWHERE

feature

  abc is
    local
      x: CLASS_X;
      proc: remote A_PROCESSOR;
    do
      ...
      create x;
      ...
      -- new processor with remote argument:
      create proc.make(x);
      ...
    end;

end -- SOMEWHERE
```

Assim, como a variável local `x` do procedimento `abc` da classe `SOMEWHERE` é passada como parâmetro onde se espera um argumento formal remoto, essa entidade passa a ser concorrente (todos os objectos à qual poderá estar associada serão também concor-

rentes).

Este exemplo – que neste aspecto de detecção de objectos concorrentes não difere da utilização de *triggers* – é um dos casos mais simples na verificação se entidades normais são concorrentes já que a entidade é uma variável local (o seu alcance restringe-se ao corpo do procedimento onde é declarada). O problema complica-se se a entidade normal for um atributo. Neste caso, esse atributo pode passar a ser uma entidade concorrente em qualquer parte do programa da respectiva classe, ou mesmo fora desta (se for público). No caso em que o atributo passa a entidade concorrente devido a ser atribuído a uma entidade remota algures no programa da própria classe – mesmo sendo um problema um pouco mais complexo do que o da variável local –, continua a ser uma decisão local à classe. A segunda situação, por outro lado, é muitíssimo mais complexa e faz com que a decisão já não possa ser tomada localmente à classe (impedindo uma compilação completa separada para cada classe), obrigando a uma análise global do programa.

À partida, não vemos nenhum razão teórica para não permitir esta última situação (razão pela qual ela é permitida na definição actual da linguagem apresentada no capítulo 6), no entanto, a sua implementação é bastante mais complexa.

Uma solução para este problema, que julgamos perfeitamente realizável, consiste em, durante a fase de compilação, gerar um grafo (dirigido) com as relações de associação relevantes entre todas as entidades com tipo do programa. Assim uma entidade normal será remota se e só se puder ser atribuída (sendo um parâmetro de um argumento formal remoto) a uma entidade remota.

No entanto, deve ser referido que actualmente a implementação do sistema de compilação do MP-Eiffel não contempla esta situação, tendo-se optado por simplificar (enormemente) este problema introduzindo uma nova anotação ao sistema de tipos, complementar ao acesso remoto, designada por visível (tipo **visible**). Assim, entidades remotas só podem estar dependentes de entidades visíveis.

A última situação resulta da possibilidade de um objecto concorrente poder dar acesso ao valor dos seus atributos e funções. O valor desses atributos ou funções é ele próprio (em linguagens orientadas a objectos puras) um objecto, pelo que esses objectos, caso não sejam completamente expandidos, terão naturalmente de também ser concorrentes. Este caso, no entanto, distingue-se dos restantes no facto de poder obrigar à partilha do sincronismo entre o objecto concorrente inicial (através do qual se obteve a referência desses outros objectos) e esses objectos.

Assim no sistema de compilação actualmente implementado uma entidade será concorrente caso seja partilhada, remota, visível ou, não sendo nenhum desses casos, se acessível através de uma entidade concorrente.

B.2.1 Grafo de dependências entre entidades

Diz-se que uma entidade de programa `x` depende de outra entidade `y`, se houver a possibilidade de `y` vir a ser atribuído a `x`. Em Eiffel essa situação só poderá ocorrer ou através das instruções de atribuição de valor (`x := ...y...` ou `x ?= ...y...`), ou se `x` for um argumento formal de uma rotina, e `y` um dos seus parâmetros actuais.

Assim, uma entidade (normal) será concorrente se, e só se, depender directa ou indirectamente, de outra entidade concorrente.

É importante referir que a dimensão e complexidade deste grafo varia na proporção da dimensão do programa, mais concretamente do respectivo número de entidades com tipo, pelo que a sua complexidade não cresce exponencialmente com o programa.

B.3 Detecção dos serviços sem efeitos colaterais

Outro dos aspectos essenciais para um implementação segura dos objectos remotos (e também da adopção de esquemas de sincronismo com menor contenção), assenta na necessidade de o sistema de compilação detectar – sem falhas – quais os serviços que não têm efeitos colaterais para o estado visível do programa[5]. A invocação de serviços remotos só será estaticamente permitida nesses casos.

Nesta perspectiva declarativa, não faz sentido permitir a invocação de procedimentos em entidades remotas (já que estes, por definição, são comandos, e como tal podem mudar o estado do programa). Existem duas possíveis excepções a esta regra, ambas a serem estudadas mais profundamente no futuro. A primeira é o caso dos serviços de execução única (principalmente as funções), já que, mesmo que tenham efeitos colaterais, estes podem não ser considerados como resultado da invocação remota, mas tão só da própria semântica desses serviços. O resultado do programa é o mesmo, independentemente do processador em particular responsável pela primeira invocação desses serviços. O segundo caso tem a ver com possibilidade de virem a existir atributos locais a cada processador (secção 5.21). Serviços que utilizem esta variedade de atributos não têm, pelo menos nesse aspecto, efeitos colaterais para a execução dos restantes processadores pelo que podem ser considerados puros no que a esse aspecto diz respeito. Aparte destas duas possíveis excepções, resta a possibilidade de invocações a atributos ou a funções.

O primeiro caso, não levanta problemas de maior, já que, novamente por definição, a observação (segura) do estado de atributos não produz efeitos colaterais,

No caso das funções, é necessário que o sistema de compilação analise apropriadamente o respectivo algoritmo, assim como o algoritmo de todos os serviços utilizados, sejam do próprio objecto ou de outros,

A simplicidade da linguagem Eiffel, ou não permitir a atribuição de valor a argumentos formais de funções (que são só de leitura), e ao deixar apenas que se atribua o valor de atributos dentro da respectiva classe, facilita tremendamente este problema. Assim as únicas instruções imperativas elementares que são responsáveis pela mudança de estado dos objectos são as instruções de atribuição de valor. E mesmo essas só serão importantes caso não se apliquem a variáveis locais (já que estas, por si só, não afectam o estado visível de nenhum objecto)

Uma vez que uma função pode invocar outras funções (incluindo ela própria), só pode haver a certeza de que uma função é pura, se o seu algoritmo não contiver atribuições de valor a atributos, e se não invocar nenhuma outra função que não seja também pura.

Em linguagens orientadas por objectos é necessário ter também em consideração a possível existência de polimorfismo subtipo e encaminhamento dinâmico (secção 3.8).

[5] O estado visível de um programa numa linguagem orientada por objectos pura, é aquele dado pelo conjunto dos estados visíveis de todos os seus objectos.

Assim, nas invocações qualificadas a rotinas tomamos a aproximação conservativa de verificar se todas as rotinas que podem ser executadas como resultado desses mecanismos são também puras. As rotinas recursivas (quer invocadas directamente na rotina ou por intermédio de outras rotinas), não colocam problemas de maior já que o sistema de compilação mantém o registo das rotinas para as quais já verificou e são puras.

B.3.1 Invocações polimórficas

Com o que já foi apresentado, é possível anotar todas as funções com tendo, ou não, efeitos colaterais. Falta no entanto, ter em consideração uma das características essenciais das linguagens orientadas por objectos: o polimorfismo e o encaminhamento dinâmico. Com efeito, sempre que é invocado um serviço, há que ter em consideração que podem, em tempo de execução, ser invocados serviços diferentes (mas com o mesmo contrato) de diferentes classes. Assim, é necessário ter em consideração todas as classes que sejam descendentes do tipo relativamente ao qual o serviço é invocado. Basta um dos serviços de uma dessas classes não ser pura para que o serviço onde a invocação é feita também não o ser.

B.3.2 Grafo de invocação de serviços

Torna-se assim necessário que o sistema de compilação crie um grafo (dirigido), cujos nós serão todos os serviços[6] de todas as classes do programa, e cujas ligações entre os nós sejam todas as invocações possíveis (incluindo, é claro, todas as invocações polimórficas). Este grafo de invocação de serviços – tal como no caso do grafo de dependências entre entidades – depende proporcionalmente da dimensão do programa, pelo que a sua complexidade é tratável.

B.4 Processadores

Neste protótipo da linguagem MP-Eiffel restringiu-se o mapeamento dos processadores a *threads* dentro de um mesmo processo num único computador. Muito embora a realização de outros mapeamentos de processadores – como por exemplo, processos no mesmo computador ou em computadores fisicamente separados – pudesse levantar problemas e condições de experimentação muito interessantes e relevantes, optou-se por dar prioridade a outros aspectos dos mecanismos. Espera-se futuramente ter condições para extender o sistema de compilação também nesse sentido.

No protótipo actual, os processadores são implementados como classes descendentes de uma classe não instanciável chamada `PROCESSOR`[7]. O sistema de compilação, sempre que há a possibilidade da criação de uma entidade remota (ou seja, criar um novo processador), gera uma nova classe descendente quer da classe `PROCESSOR` (o construtor utilizado será implementado como a redefinição do programa do processador), quer da classe associada à entidade.

A figura B.2 exemplifica esta situação.

[6] Bastam os "vivos". Ou seja, aqueles que podem ser utilizados em tempo de execução pelo programa.
[7] O código fonte pode ser consultado no apêndice E.1.

```
class CLASS_X

creation
  make

...

end -- CLASS_X
```

```
class CLASS_X_PROCESSOR

inherit
  CLASS_X;
  PROCESSOR
    rename
      main as make
    end

end -- CLASS_X_PROCESSOR
```

Figura B.2: Realização de processadores.

B.4.1 Detecção do fim do programa

Um programa em MP-Eiffel estará terminado quando nenhum dos seus processadores estiver em execução (ou seja, ou já terminou, ou está num estado de espera por *triggers*.

Este comportamento foi implementado na própria classe associada aos processadores. O fim do programa é detectado verificando a ocorrência de duas condições simultâneas:

- se o número de processadores em espera é igual ao número de processadores existentes;
- e se todas as filas de mensagens de *triggers* associadas a cada processador estão vazias.

Nenhuma das duas condições separadamente é suficiente para garantir a total inactividade de todos os processadores do programa. Pode acontecer que o número de processadores seja temporariamente igual ao número de processadores em espera, havendo ainda *triggers* para executar (já que é o próprio processador que incrementa o contador de processadores em espera, e nesse intervalo podem-lhe enviar um novo *trigger*). E também, evidentemente, a não existência num determinado instante de *triggers* não invalida a possibilidade de existirem processadores a executarem os respectivos programas.

B.5 *Triggers*

A implementação deste mecanismo, como era aliás esperado, mostrou ser bastante mais simples do que os mecanismos de comunicação entre processadores por partilha de memória.

Nesta implementação teve-se de ter em conta os seguintes aspectos deste mecanismo:

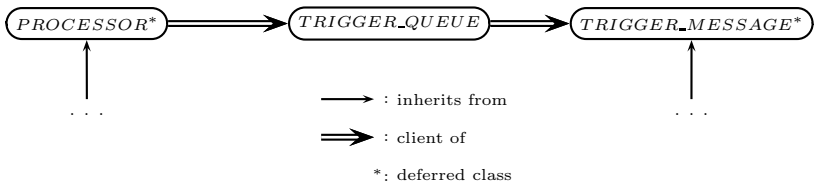

Figura B.3: Implementação de *triggers*.

- Os *triggers* têm semânticas diferentes consoante estão associados a procedimentos ou a outros serviços, tendo um comportamento, respectivamente, assíncrono ou síncrono. Este aspecto afecta não só o código a associar ao programa no lado dos emissores, como também o dos receptores, já que as excepções comportam-se de forma bastante diferente nos dois casos (ver secção 5.18).

- Qualquer serviço pode vir a estar associado a um *trigger* pelo que o mecanismo tem de ter em conta a diversidade imensa entre esses serviços. Em particular, é conveniente, quando for necessário, arranjar uma forma eficiente de passar os argumentos para os *triggers*.

- O comportamento das pré-condições concorrentes tem de ser tido em conta também neste mecanismo.

- Uma falha numa pré-condição sequencial terá de ser devidamente propagada para o processador emissor do respectivo *trigger*.

Um outro aspecto também tido em conta embora seja relativamente pouco importante, resulta do facto de um processador só poder receber *triggers* se criar pelo menos um objecto que os declare (e também – embora este aspecto não tenha sido considerado – se o programa do processador disponibilizar as referências desses objectos para acessos remotos).

A figura B.3 mostra a estrutura básica das classes desenvolvidas para a geração de código EIFFEL de suporte em tempo de execução para implementação deste mecanismo[8].

A cada *trigger* diferente, o sistema de compilação cria uma nova classe descendente da classe `TRIGGER_MESSAGE` onde todos os aspectos importantes para a posterior execução do *trigger* – a saber, a ligação ao serviço associado ao trigger, a passagem de eventuais argumentos, e a identificação do processador emissor (sem a qual não seria possível propagar possíveis excepções) – são encapsulados. A classe `TRIGGER_MESSAGE` (secção E.2) contém o TDA comum a todos os *triggers* e suficiente para a sua execução polimórfica na classe `PROCESSOR` (secção E.1).

A cada processador (instância da classe `PROCESSOR`) estará associada uma instância da classe `TRIGGER_QUEUE` (secção E.3), que implementa uma fila *FIFO* de *triggers*.

[8] O código fonte dessas classes pode ser encontrado no apêndice E.

Apêndice C

Implementação de esquemas de sincronismo

C.1 Exemplos de realização de esquemas de sincronismo simples

O código apresentado aqui é EIFFEL puro testável e foi compilado com a versão segura THREAD-SAFE SMALLEIFFEL (apêndice D).

C.1.1 Stack

```
-- Generic unbounded STACK class

deferred class STACK[E]

feature

  count: INTEGER is
      -- Number of elements
    deferred
    end;

  empty: BOOLEAN is
    do
      Result := count = 0
    end;

  top: E is
      -- STACK's last pushed element
    require
      not empty
    deferred
    ensure
      same_count: count = old count
    end;

  push(elem: like top) is
    deferred
    ensure
      one_more: count = old count + 1;
      element_placed_on_top: top = elem;
    end;

  pop is
    require
      not empty
    deferred
    ensure
      one_less: count = old count - 1
    end;

invariant

  count >= 0;
  empty = (count = 0)

end -- STACK
```

C.1.2 Stack: Monitor

```
class MONITOR_STACK[E]

  creation
    make

  feature {NONE}

    stack: STACK[E];
    mtx: MUTEX;
    cnd_var: CONDITION_VARIABLE;

  feature

    make(s: STACK[E]) is
      require
        s /= Void
      do
        stack := s;
        create mtx.make;
        create cnd_var.make
      end;

  feature

    count: INTEGER is
      do
        mtx.lock;
        Result := stack.count;
        mtx.unlock
      end;

    empty: BOOLEAN is
      do
        mtx.lock;
        Result := stack.empty;
        mtx.unlock
      end;

    top: E is
      do
        mtx.lock;
        from until not empty loop
          cnd_var.wait(mtx)
        end;
        Result := stack.top;
        mtx.unlock
      end;

    push(elem: like top) is
      do
        mtx.lock;
        stack.push(elem);
        mtx.unlock;
        cnd_var.broadcast
      end;

    pop is
      do
        mtx.lock;
        from until not empty loop
          cnd_var.wait(mtx)
        end;
        stack.pop;
        mtx.unlock;
        cnd_var.broadcast
      end;

end -- MONITOR_STACK
```

C.1.3 Stack: Exclusão Leitores-Escritor

```
class RW_EXCLUSION_STACK[E]

  creation
    make

  feature {NONE}

    stack: STACK[E];
    rwl: READ_WRITE_LOCK;
    mtx: MUTEX;
    cnd_var: CONDITION_VARIABLE;

  feature

    make(s: STACK[E]) is
      require
        s /= Void
      do
        stack := s;
        create rwl.make;
        create mtx.make;
        create cnd_var.make
      end;

  feature

    count: INTEGER is
      do
```

```
      rwl.read_lock;
      Result := stack.count;
      rwl.read_unlock
    end;

  empty: BOOLEAN is
    do
      rwl.read_lock;
      Result := stack.empty;
      rwl.read_unlock
    end;

  top: E is
    do
      rwl.read_lock;
      from until not empty loop
        rwl.read_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.read_lock;
      end;
      Result := stack.top;
      rwl.read_unlock
    end;

  push(elem: like top) is
    do
      rwl.write_lock;
      stack.push(elem);
      rwl.write_unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      rwl.write_lock;
      from until not empty loop
        rwl.write_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.write_lock;
      end;
      stack.pop;
      rwl.write_unlock;
      cnd_var.broadcast
    end;

end -- RW_EXCLUSION_STACK
```

C.1.4 Stack: Leitores-Escritor Concurrentes (Lamport)

```
class RW_CONCURRENT_LAMPORT_STACK[E]

  creation
    make

  feature {NONE}

    stack: STACK[E];
    mtx: MUTEX;
    writer_in,writer_out: INTEGER;
    cnd_var: CONDITION_VARIABLE;

  feature

    make(s: STACK[E]) is
      require
        s /= Void
      do
        stack := s;
        create mtx.make;
        create cnd_var.make
      end;

  feature

    count: INTEGER is
      local
        success: BOOLEAN;
        v: INTEGER
      do
        from until success loop
          v := writer_in;
          Result := stack.count;
          success := v = writer_out
        end;
      rescue
        if v /= writer_out then
          retry
        end
      end;

    empty: BOOLEAN is
      local
        success: BOOLEAN;
        v: INTEGER
      do
        from until success loop
          v := writer_in;
          Result := stack.empty;
          success := v = writer_out
        end;
      rescue
        if v /= writer_out then
          retry
```

```
    end
  end;

top: E is
  local
    success: BOOLEAN;
    v: INTEGER
  do
    from until success loop
      v := writer_in;
      from until not empty loop
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
      end;
      Result := stack.top;
      success := v = writer_out;
    end;
  rescue
    if v /= writer_out then
      retry
    end
  end;

push(elem: like top) is
  do
    mtx.lock;
    writer_in := writer_in + 1;
    stack.push(elem);
    writer_out := writer_out + 1;
    mtx.unlock;
    cnd_var.broadcast
  end;

pop is
  do
    mtx.lock;
    from until not empty loop
      cnd_var.wait(mtx)
    end;
    writer_in := writer_in + 1;
    stack.pop;
    writer_out := writer_out + 1;
    mtx.unlock;
    cnd_var.broadcast
  end;

end -- RW_CONCURRENT_LAMPORT_STACK
```

C.2 Exemplo de algoritmos sem bloqueamento

Os algoritmos genéricos para este tipo de sincronismo assentam basicamente em três fases: é retirada uma cópia (estável) do estado do objecto; aplica-se a operação desejada a essa cópia; e por fim, caso o objecto não tenha sido modificado desde a cópia feita, substitui-se atomicamente o estado actual do objecto por essa cópia modificada. O processo é repetido até que seja bem sucedido.

No caso (desejável) de se separar os serviços dos objectos em comandos e consultas, podemos simplificar bastante o algoritmo aplicável às últimas. Com efeito, para estes, não é necessária a substituição atómica do estado do objecto, bastando, para que a operação seja bem sucedida, garantir que esta é aplicada a uma cópia válida (estável) do objecto.

Algoritmo Sem Bloqueamento Genérico para Comandos

```
1.  fail = true;
2.  do
    {
3.    obj_cpy.copy(obj);
4.    if (obj_cpy.copy_succeed(obj))
      {
5.      obj_cpy.command(...);
6.      fail = !obj.atomic_replace_on_linearizability(obj_cpy);
      }
    }
7.  while(fail);
```

Algoritmo Sem Bloqueamento Genérico para Consultas

```
1.  fail = true;
2.  do
    {
3.    obj_cpy.copy(obj);
4.    if (obj_cpy.copy_succeed(obj))
      {
5.      result = obj_cpy.query(...);
6.      fail = false;
      }
    }
7.  while(fail);
```

Os algoritmos anteriores, em pseudo-código tipo C++, exemplificam possíveis aproximações à sincronização automática com esquema de sincronismo. Em ambos os casos, é feita uma cópia de `obj` para `obj_cpy` (`3.`), após o que, caso esta tenha sido bem sucedida (`4.`), é invocada a operação desejada utilizando a cópia do objecto (`5.`). No caso dos comandos, e caso a linearizabilidade se verifique, substitui-se o estado de `obj` pelo o de `obj_cpy` (`6.`). Se esta substituição não for possível, todo o processo é repetido (`7.`).

C.3 Verificação do invariante em esquemas mistos de sincronismo com concorrência

C.3.1 Implementação da verificação do invariante

```
#include <pthread.h>

typedef struct
{
  int counter;
  int done_start;
  int Result_start;
  int Result_end;
  pthread_mutex_t mtx;
  pthread_cond_t cnd;
} INVARIANT_SYNCH;
#define INVARIANT_SYNCH_INIT \
```

```
  {0,0,0,0,PTHREAD_MUTEX_INITIALIZER,PTHREAD_COND_INITIALIZER}

int command_test_invariant(int (*inv)(void *obj),void *obj,
                           INVARIANT_SYNCH *synch,int start_of_routine)
{
  int Result;

  pthread_mutex_lock(&synch->mtx);
  if (start_of_routine)
  {
    synch->counter++;
    if (!synch->done_start)
    {
        // Invariant checked only in the first routine
        // (except for creation command, instead of rechecking
        //  the invariant, we could reuse the last Result_end).
      synch->Result_start = (*inv)(obj);
      synch->done_start = 1;
    }
      // Invariant result reused for all concurrent routines
    Result = synch->Result_start;
  }
  else // end_of_routine
  {
    synch->counter--;
    if (synch->counter == 0)
    {
        // Invariant checked only in the last routine
      synch->done_start = 0;
      synch->Result_end = (*inv)(obj);
        // awake all waiting processors (barrier end)
      pthread_cond_broadcast(&synch->cnd);
    }
    else
    {
        // wait for the last routine
      while(synch->counter > 0)
        pthread_cond_wait(&synch->cnd,&synch->mtx);
    }
    Result = synch->Result_end;
  }
  pthread_mutex_unlock(&synch->mtx);

  return Result;
}

int query_test_invariant(int (*inv)(void *obj),void *obj,
                         INVARIANT_SYNCH *synch)
{
  int Result;

  pthread_mutex_lock(&synch->mtx);
    // fetch last invariant verification
  if (synch->done_start)
    Result = synch->Result_start;
  else
    Result = synch->Result_end;
```

```
    pthread_mutex_unlock(&synch->mtx);

    return Result;
  }
```

C.3.2 Implementação de serviços tipo consulta (pura)

```
1.    if (!query_test_invariant(...))
1.1.    raise_invariant_exception(...);
2.    if (!test_precondition(...))
2.1.    raise_precondition_exception(...);
3.    Result = execute_query_body(...);
4.    if (!test_postcondition(...))
4.1.    raise_postcondition_exception(...);
5.    if (!query_test_invariant(...))
5.1.    raise_invariant_exception(...);
```

C.3.3 Implementação de serviços tipo comando

```
1.    if (!command_test_invariant(... ,1))
1.1.    raise_invariant_exception(...);
2.    if (!test_precondition(...))
2.1.    raise_precondition_exception(...);
3.    execute_command_body(...);
4.    if (!test_postcondition(...))
4.1.    raise_postcondition_exception(...);
5.    if (!command_test_invariant(... ,0))
5.1.    raise_invariant_exception(...);
```

Apêndice D

Thread-Safe SmallEiffel

Na concepção do compilador de MP-Eiffel optou-se por utilizar uma ferramenta de geração de "parsers"e "scanners": PCCTS, e implementar todo o código em Eiffel. Para que tal fosse possível era necessário que o compilador de Eiffel utilizado – SmallEiffel – gerasse código thread-safe, pelo que foi preciso alterar o próprio compilador SmallEiffel.

Assim no âmbito deste trabalho, fez-se uma versão thread-safe de SmallEiffel (que foi colocada no domínio público), conjuntamente com uma biblioteca de classes de manipulação de threads.

Esta biblioteca é composta pelas seguintes classes:

- THREAD
- THREAD_CONTROL
- THREAD_ID
- MUTEX
- CONDITION_VARIABLE
- READ_WRITE_LOCK
- ONCE_MANAGER
- THREAD_BARRIER
- THREAD_PIPELINE
- THREAD_ATTRIBUTE
- GROUP_MUTEX

D.1 Classe THREAD

```
deferred class THREAD

inherit
  THREAD_CONTROL

feature {THREAD}

  main
      -- New thread starting point (main routine).
      -- Is not called directly, but in 'start*' routines
      -- The new thread start object will be 'Current'
      -- The thread terminates at the end of 'main'
    deferred
    end;

feature

  start
      -- start new thread
    require
      not is_expanded_type

  start_detached
      -- start new thread on detached (unjoinable) state

  start_with_name(n: STRING)
      -- start new thread named 'n'
    require
      not is_expanded_type;
      n /= Void

  start_detached_with_name(n: STRING)
      -- start new thread on detached (unjoinable) state,
      -- named 'n'
    require
      n /= Void

feature

  my_birth_id: THREAD_ID;

end -- THREAD
```

D.2 Classe THREAD_CONTROL

```
class THREAD_CONTROL

feature

  running: BOOLEAN

  detached: BOOLEAN

  is_same_thread(other: THREAD): BOOLEAN
      -- is the calling thread the same as the
      -- owner of 'other'?

  is_main_thread,is_root_thread: BOOLEAN
      -- are we in main (root) thread?

  thread_name_defined: BOOLEAN

  thread_name: STRING
    require
      thread_name_defined

  set_thread_name(n: STRING)
    require
      n /= Void

  detach
      -- detach current thread
    require
      running;
      not detached

  exit
      -- forces termination of current thread
    require
      running

  join(other: THREAD)
      -- The caller will block while 'other' thread is running
    require
      not other.detached;
      not other.running or else not is_same_thread(other)

  join_all_childs,join_all
      -- The caller will block while all direct child threads
      --  of the owner of current object are running
      -- Ignores detached direct childs.
      -- This feature is usable by the thread owning Current
      --  object (unlike 'join' feature).
      -- Returns immediately if there isn't any child.

feature

  thread_id: THREAD_ID

end -- THREAD_CONTROL
```

D.3 Classe THREAD_ID

```
class THREAD_ID

inherit
  THREAD_CONTROL

creation
  make

feature

  make
      -- fetches the id of the creation thread!

  same_as(other: like Current): BOOLEAN
    require
      other /= Void

end -- THREAD_ID
```

D.4 Classe MUTEX

```
class MUTEX

 creation
   make

 feature

   initialized: BOOLEAN

   make

   destroy
     -- destroys mutex

   lock

   try_lock: BOOLEAN
     -- on lock success returns true (false otherwise)

   unlock

end -- MUTEX
```

D.5 Classe CONDITION_VARIABLE

```
class CONDITION_VARIABLE

 creation
   make

 feature

   initialized: BOOLEAN

   make

   destroy
     -- destroys condition variable

   wait(m: MUTEX)
       -- m must be locked

   timedwait(m: MUTEX;timeout: INTEGER): BOOLEAN
     -- Returns false on timeout, and true if signaled
     --  timeout is the absolute time in seconds (relative
     --  to 00:00:00 GMT, January 1, 1970)
     -- Absolute time is used, instead of elapsed time,
     --  because of spurious wakenings (always possible
     --  with cond. variables).

   signal

   broadcast

end -- CONDITION_VARIABLE
```

D.6 Classe READ_WRITE_LOCK

```
class READ_WRITE_LOCK

 creation
   make,make_with_write_priority,make_with_read_priority

 feature

   make,make_with_write_priority

   make_with_read_priority

   destroy

   read_lock

   read_try_lock: BOOLEAN
     -- on lock success returns true (false otherwise)

   read_unlock

   write_lock

   write_try_lock: BOOLEAN
     -- on lock success returns true (false otherwise)

   write_unlock

   write_lock_priority: BOOLEAN

   read_lock_priority: BOOLEAN

end -- READ_WRITE_LOCK
```

D.7 Classe ONCE_MANAGER

```
expanded class ONCE_MANAGER

 feature

   refresh(key: STRING)
     require
       key /= Void

   refresh_some(key_list: ARRAY[STRING])
     require
       key_list /= Void

   refresh_all

end -- ONCE_MANAGER
```

D.8 Classe THREAD_BARRIER

```
class THREAD_BARRIER

creation
  make,make_static

feature

  make

  make_static(size: INTEGER)
    require
      size > 0

  terminated: BOOLEAN

  terminate

  release
      -- all waiting threads in barrier will be released.

  is_static: BOOLEAN
    -- is the size of the barrier fixed?

  set_number_of_threads(size: INTEGER)
    require
      is_static;
      size > 0;

  number_of_threads: INTEGER
      -- number of signed threads

  signed: BOOLEAN
      -- is calling thread already signed?
    require
      not is_static

  sign_on
      -- calling thread will be a new user of barrier
    require
      not is_static;
      not signed

  sign_off
      -- calling thread won't be a user of barrier anymore
    require
      not is_static;
      signed

  wait
      -- Calling thread will wait until 'number_of_threads'
      --  threads are waiting (then they will all unblock).
      -- On termination initializes new barrier (with the
      --  same threads if the barrier is dynamic)
    require
      is_static or else signed

end -- THREAD_BARRIER
```

D.9 Classe THREAD_PIPELINE

```
class THREAD_PIPELINE

inherit
  THREAD_CONTROL

creation
  make

feature

  make

  add_thread(thr: THREAD)
      -- adds a new concurrent thread to current [last] "pipe".
    require
      thr /= Void

  empty_pipe: BOOLEAN
      -- is current pipe empty?

  new_pipe
      -- appends a new empty "pipe" to pipeline.
    require
      current_pipe_not_empty: not empty_pipe

  start
      -- starts pipeline thread execution.
      -- exits only on pipeline termination.

end -- THREAD_PIPELINE
```

D.10 Classe THREAD_ATTRIBUTE

```
expanded class THREAD_ATTRIBUTE[T]

feature

  put(e: T)

  item: T

end -- THREAD_ATTRIBUTE
```

D.11 Classe GROUP_MUTEX

```
class GROUP_MUTEX

creation
  make

feature

  make(num_groups: INTEGER)
    require
      num_groups >= 2

  destroy
```

```
  number_of_groups: INTEGER

  lock(g: INTEGER)
    require
      g >= 1 and g <= number_of_groups

  try_lock(g: INTEGER): BOOLEAN
    -- on lock success returns true (false otherwise)
    require
      g >= 1 and g <= number_of_groups

  unlock(g: INTEGER)
    require
      g >= 1 and g <= number_of_groups

feature
  -- group priorities (default is by the number of the group,
  --  from the highest priority [group 1] to the lowest
  --  [group number_of_groups].

  highest_priority_group: INTEGER

  lowest_priority_group: INTEGER

  greater_than_group_priority(g1,g2:INTEGER): BOOLEAN
      -- priority(g1) > priority(g2) ?
    require
      g1 /= g2;
      g1 >= 1 and g1 <= number_of_groups;
      g2 >= 1 and g2 <= number_of_groups

  lower_than_group_priority(g1,g2:INTEGER): BOOLEAN
      -- priority(g1) < priority(g2) ?
    require
      g1 /= g2;
      g1 >= 1 and g1 <= number_of_groups;
      g2 >= 1 and g2 <= number_of_groups

  set_highest_priority(g: INTEGER)
      -- moves group g to highest priority (other groups
      --  maintain their relative ordering)
    require
      g >= 1 and g <= number_of_groups

  set_lowest_priority(g: INTEGER)
      -- moves group g to lowest priority (other groups
      --  maintain their relative ordering)
    require
      g >= 1 and g <= number_of_groups

  increase_group_priority(g: INTEGER)
    require
      (g >= 1 and g <= number_of_groups) and then
      g /= highest_priority_group

  decrease_group_priority(g: INTEGER)
    require
      (g >= 1 and g <= number_of_groups) and then
      g /= lowest_priority_group

  set_default_priorities

  print_priority_lock_list

invariant

  number_of_groups >= 2

end -- GROUP_MUTEX
```

Apêndice E

Algumas classes de suporte à compilação de MP-Eiffel

E.1 Classe PROCESSOR

```
deferred class PROCESSOR

inherit
  THREAD
    rename
      main as life
    end;

feature -- PROCESSOR main program

  main is
    deferred
    end;

feature

  life is
      -- processor (boring) life
      -- detection of no program activity not optimized!
    local
      msg: TRIGGER_MESSAGE
    do
      !!cnd_var.make;
      register_processor(Current);
      main;
      increment_waiting_processors;
      if program_with_no_activity then
        terminate_program
      else
        if not triggers_enabled then
            -- triggers might became enabled due to a
            -- sequential precondition failure response
            -- to a asynchronous trigger call
          mtx.lock;
          cnd_var.wait(mtx);
          mtx.unlock
        end;
        if triggers_enabled then
          from until trigger_queue.is_terminated loop
            msg := trigger_queue.fetch_trigger;
            decrement_waiting_processors;
            msg.execute_call;
            increment_waiting_processors;
            if program_with_no_activity then
              terminate_program
            end
          end
        end
      end
    end

feature
  -- exception (to be used when a trigger call is executed)
  precondition_failed: BOOLEAN;

  notify_precondition_failure is
    do
      precondition_failed := true
    end;

  reset_precondition_failure is
    do
      precondition_failed := false
    end;

feature -- triggers

  enable_triggers is
      -- to be called during main execution if an object
      --  with triggers is created by the processor.
    once {"object","processor"}
      global_mutex.lock;
      !!trigger_queue.make;
      triggers_enabled := true;
      cnd_var.signal;
      global_mutex.unlock
    end;

feature {NONE} -- triggers

  triggers_enabled: BOOLEAN; -- default is false

  trigger_queue: TRIGGER_QUEUE;

  mtx: MUTEX;

  cnd_var: CONDITION_VARIABLE;

feature {NONE} -- features shared by all processors!

  global_mutex: MUTEX is
    once {"class","program"}
      !!Result.make
    end;

  waiting_proc_ref: INTEGER_REF is
    once {"class","program"}
      !!Result
    end;

  waiting_processors: INTEGER is
    do
      global_mutex.lock;
      Result := waiting_proc_ref.item;
      global_mutex.unlock
    end;
```

```
  unlocked_increment_waiting_processors is
    do
      waiting_proc_ref.set_item(waiting_proc_ref.item+1);
      check
        waiting_proc_ref.item <= unlocked_number_of_processors
      end
    end;

  unlocked_decrement_waiting_processors is
    do
      waiting_proc_ref.set_item(waiting_proc_ref.item-1);
      check waiting_proc_ref.item >= 0 end
    end;

  increment_waiting_processors is
    do
      global_mutex.lock;
      unlocked_increment_waiting_processors;
      global_mutex.unlock
    end;

  decrement_waiting_processors is
    do
      global_mutex.lock;
      unlocked_decrement_waiting_processors;
      global_mutex.unlock
    end;

  terminate_program is
    do
      global_mutex.lock;
      from
        all_processors.start
      until
        all_processors.off
      loop
        if all_processors.item.triggers_enabled then
          all_processors.item.trigger_queue.terminate
        end;
        all_processors.item.cnd_var.signal;
        all_processors.forth
      end;
      global_mutex.unlock;
    end;

  program_with_no_activity: BOOLEAN is
      -- all trigger's queues empty and all processors waiting
    do
      global_mutex.lock;
      if unlocked_number_of_processors = waiting_proc_ref.item then
        from
          all_processors.start
        until
          all_processors.off or else
          (all_processors.item.triggers_enabled and then
           not all_processors.item.trigger_queue.is_empty)
        loop
          all_processors.forth
        end;
        Result := all_processors.off
      end;
      global_mutex.unlock;
    end;

  all_processors: DYNAMIC_LIST[PROCESSOR] is
    local
      factory: DYNAMIC_LIST_FACTORY[PROCESSOR]
    once {"class","program"}
      global_mutex.lock;
      !!factory;
      Result := factory.make_dynamic_list;
      global_mutex.unlock
    end;

  register_processor(p: PROCESSOR) is
    do
      global_mutex.lock;
      all_processors.append(p);
      global_mutex.unlock
    end;

  unlocked_number_of_processors: INTEGER is
    do
      Result := all_processors.count
    end;

end -- PROCESSOR
```

E.2 Classe TRIGGER_MESSAGE

```
deferred class TRIGGER_MESSAGE
--    A new class is created by the compiling system for
--  each possible trigger message. That class will include
--  all the required actual arguments necessary to execute
--  the call (actual_call). The compiling system implements
--  appropriately the deferred routines.

feature

  actual_call is
    deferred
    end;

  execute_call is
    local
      precond_fail: SEQUENTIAL_PRECONDITION_FAILURE
    do
      if not sequential_precondition then
        if is_synchronous then
            -- precondition failure is propagated to the caller,
            -- without affecting the callee
          caller.notify_precondition_failure
        else
          !!precond_fail;
          caller.enable_triggers;
          caller.trigger_queue.enqueue_trigger(precond_fail)
        end
      else
        wait_for_concurrent_precondition;
        actual_call
      end;
    end

  is_asynchronous: BOOLEAN is
      -- true is procedure call
      -- (redefined to the appropriate constant boolean value)
    deferred
    end;

  is_synchronous: BOOLEAN is
      -- true is valued feature call
      -- (redefined to the appropriate constant boolean value)
    deferred
    end;

  sequential_precondition: BOOLEAN is
    deferred
    end;

  wait_for_concurrent_precondition is
    deferred
    end;

  caller: PROCESSOR;

  set_caller(p: PROCESSOR) is
    do
      caller := p
    end;

end -- TRIGGER_MESSAGE
```

E.3 Classe TRIGGER_QUEUE

```
class TRIGGER_QUEUE

creation
  make

feature

  make is
    local
      factory: QUEUE_FACTORY[TRIGGER_MESSAGE];
    do
      !!mtx.make;
      !!cnd_var.make;
      !!factory;
      queue := factory.make_queue
    end;

  enqueue_trigger(tm: TRIGGER_MESSAGE) is
    require
      tm /= Void
    do
      mtx.lock;
      queue.enqueue(tm);
      cnd_var.signal;
      mtx.unlock;
    end;

  fetch_trigger: TRIGGER_MESSAGE is
    do
      mtx.lock;
      from until terminated or else not queue.empty loop
        cnd_var.wait(mtx);
      end;
      if not terminated then
        Result := queue.tail;
        queue.dequeue;
      end;
      mtx.unlock;
    end;

  is_empty: BOOLEAN is
    do
      mtx.lock;
      Result := queue.empty;
      mtx.unlock
    end;

  is_terminated: BOOLEAN is
    do
      mtx.lock;
      Result := terminated;
      mtx.unlock
    end;

  terminate is
    do
      mtx.lock;
      terminated := true;
      cnd_var.signal;
      mtx.unlock
    end;

feature {NONE}

  terminated: BOOLEAN;

  mtx: MUTEX;

  cnd_var: CONDITION_VARIABLE;

  queue: QUEUE[TRIGGER_MESSAGE];

end -- TRIGGER_QUEUE
```

E.4 Classe SEQUENTIAL_PRECONDITION_FAILURE

```
class SEQUENTIAL_PRECONDITION_FAILURE

inherit
  TRIGGER_MESSAGE

feature

  actual_call is
    require
      false
    do
    end;

  is_asynchronous: BOOLEAN is true;

  is_synchronous: BOOLEAN is false;

  sequential_precondition: BOOLEAN is true;

  wait_for_concurrent_precondition is
    do
    end;

end -- SEQUENTIAL_PRECONDITION_FAILURE
```

Glossário

Asserção [*assertion*]: Condição booleana (predicado) a ser verificada nesse ponto do programa por forma a que este não esteja incorrecto.

Asserção concorrente: Asserção com uma condição concorrente.

Asserção de classe: Invariantes, pré-condições e pós-condições.

Asserção formal: Parte de uma asserção que pode ser executada pelo programa.

Asserção informal: Parte de uma asserção não executável pelo programa.

Atributo: Registo de informação pertencente a objectos.

Colector de lixo: Método de gestão automática de memória.

Comando [*command*]: Serviço de modificação do objecto (procedimento).

Condição concorrente: Predicado que pode depender de outro processador que não o que o está a testar.

Consulta [*query*]: Serviço de observação do objecto (função ou atributo).

Entidades com tipo [*typed entities*]: Elementos sintácticos de uma linguagem que estão associados a um "tipo".

Escalonamento [*scheduling*]: Estratégia para seleccionar os processadores a executar.

Threads: Unidades de processamento concorrente baseadas na partilha de memória e de outros recursos do sistema operativo entre elas. São caracterizadas por minimizarem a troca de contexto requerida para o escalonamento de diferentes *threads* e de fazerem parte de um único processo do sistema operativo.

Instruções estruturadas puras: Instruções cuja semântica é definida explicitamente de "fora-para-dentro". Permitem a a composição e decomposição de algoritmos por blocos encaixados.

Linguagens imperativas: Linguagens cujos algoritmo são expressos como uma sequência de comandos que podem modificar explicitamente o estado do sistema.

Linguagens orientadas por objectos puras: Linguagens cujos programas são compostos apenas por objectos.

Método: Rotina.

Objecto concorrente: Objecto utilizável por mais do que um processador.

Polimorfismo de subtipo (de inclusão): Mecanismo que permite que se associe objectos a uma entidade, desde que os tipos dos objectos sejam subtipos do tipo da entidade.

Polimorfismo paramétrico: Mecanismo que permite a especificação de classes em função de tipos genéricos.

Polimorfismo *ad-hoc*: Mecanismo que permite a definição de diferentes serviços com o mesmo nome, desde que tenham uma assinatura estática diferente.

Processador abstracto: Noção abstracta de processador sem ligação a nenhum suporte específico de execução.

Processador escritor: Processador enquanto executa comandos ou consultas impuras.

Processador leitor: Processador enquanto executa consultas puras.

Processador [*processor*]: unidade de processamento autónoma capaz de suportar a execução sequencial de instruções.

Processamento heterogéneo: Quando os processadores podem estar associados a diferentes suportes de execução.

Processamento homogéneo: Quando os processadores só podem estar associados a um suporte de execução.

Processo [*process*]: Unidade de processamento concorrente de sistemas operativos. São caracterizadas por terem uma baixa coesão entre diferentes processos (ao contrário das *threads*).

Rotina [*routine*]: Função ou procedimento de uma classe.

Serviço abstracto: Serviço sem implementação (apenas representado pela sua interface).

Serviço de classe: Serviço partilhado por todas as instâncias de uma classe.

Serviço de execução única: Serviços executados apenas a primeira vez que são invocados.

Serviço [*feature*]: Rotina ou atributo de uma classe.

Sincronismo condicional: Sincronismo que condiciona o uso de objectos à verificação de determinadas condições.

Sincronismo inter-objecto: Sincronismo que permite vários usos exclusivos de um ou mais objectos concorrentes.

Sincronismo intra-objecto: Sincronismo que protege os serviços internos de um objecto concorrente uns dos outros.

Sistema de suporte à execução de programas: O conjunto formado pelo *hardware* e o(s) sistema(s) operativo(s) do sistema de computação onde o programa é executado.

Sistemas de programação concorrente: Sistemas que suportam a programação concorrente, seja por intermédio de bibliotecas de *software*, de linguagens concorrentes, ou por uma mistura de ambas.

SMP [*Symmetric MultiProcessing*]: Arquitectura de computadores baseada em múltiplas unidades de processamento central a operar com partilha de memória.

Subclasse: Classe descendente.

Subtipo: Uma classe `A` é subtipo de uma classe `B`, se as instâncias de `A` puderem ser utilizadas em entidades do tipo `B`.

Super-classe: Classe ascendente.

Super-tipo: Relação inversa do subtipo.

TDA [*ADT (Abstract Data Type)*]: Tipo de Dados Abstracto.

Referências bibliográficas

[Ada95 95] *Ada 95 Reference Manual (Language and Standard Libraries).* U.S. Government, 1995. 3.8.1, 3.19, 4.5.2, 4.5.3, 5.4.2, 5.10.10

[Agha 86] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Massachusetts, 1986. 5.4.3

[Agha 99] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing", *Journal of Systems Architecture*, 45(15), September 1999. 5.4.3

[America 87a] P. America, "Inheritance and subtyping in a parallel object-oriented language". In *European conference on object-oriented programming on ECOOP '87*, pages 234–242, Springer-Verlag, London, UK, 1987. 5.16

[America 87b] P. America, "Pool-t: A parallel object-oriented language". In A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming*, pages 199–220, MIT Press, 1987. 5.4.2

[Anderson 97] J. H. Anderson, R. Jain, and S. Ramamurthy, "Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors". In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 111–122, December 1997. 5.10.6

[Andrews 83] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming", *ACM Comput. Surv.*, 15(1):3–43, 1983. 1, 4.3, 10, 4.5, 4.5.2, 4.5.2, 4.6

[Arslan 06] V. Arslan and B. Meyer, "Asynchronous exceptions in concurrent object-oriented programming". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 62–70, University of York – Department of Computer Science, July 2006. 5.18

[Baquero 95] C. Baquero, R. Oliveira, and F. Moura, "Integration of concurrency control in a language with subtyping and subclassing". In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS'95)*, pages 173–184, USENIX Association, June 1995. 5.16

[BH 72] P. Brinch Hansen, "Structured multiprogramming", *Communications of the ACM*, 15(7):574–578, 1972. 4.6.4

[BH 73] P. Brinch Hansen, *Operating System Principles.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973. 6.1

[BH 75] P. Brinch Hansen, "The programming language concurrent pascal.", *IEEE Trans. Software Eng.*, 1(2):199–207, 1975. 4.1.2, 4.5.3

[BH 93] P. Brinch Hansen, "Monitors and concurrent pascal: a personal history". In *The second ACM SIGPLAN conference on History of programming languages*, pages 1–35, ACM Press, 1993. 5.10.3

[BH 99] P. Brinch Hansen, "Java's insecure parallelism", *ACM SIGPLAN Notices*, 34(4):38–45, 1999. 5.10.2, 5.10.3

[Bobrow 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon, "Common lisp object system specification", *SIGPLAN Not.*, 23(SI):1–142, 1988. 3.5

[Borning 86] A. H. Borning, "Classes versus prototypes in object-oriented languages". In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 36–40, IEEE Computer Society Press, Los Alamitos, CA, USA, 1986. 3.5

[Briot 87] J.-P. Briot and A. Yonezawa, "Inheritance and synchronization in concurrent oop". In *European conference on object-oriented programming on ECOOP '87*, Springer-Verlag, London, UK, 1987. 5.16

[Briot 98] J.-P. Briot, R. Guerraoui, and K.-P. Lohr, "Concurrency and distribution in object-oriented programming", *ACM Computing Surveys (CSUR)*, 30(3):291–329, 1998. 5.9

[Bruce 02] K. B. Bruce, *Foundations of Object-Oriented Languages – Types and Semantics*. The MIT Press, Cambridge, Massachusetts, 2002. 3.1, 3.8.2, 3.8.3, 3.8.4

[Bruce 93] K. B. Bruce, "Safe type checking in a statically-typed object-oriented programming language". In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, ACM Press, 1993. 3.8.3

[Butenhof 97] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley, 1997. 4.1.2, 4.5.3, 5.21

[Böhm 66] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules", *Communications of the ACM*, 9(5):366–371, 1966. 7

[Canning 89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell, "F-bounded polymorphism for object-oriented programming". In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, ACM Press, New York, NY, USA, 1989. 3.10.2

[Cardelli 85] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism", *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985. 3.8, 3.8.2, 3.10, 3.10.2, 3.14

[Cardelli 88] L. Cardelli, "Structural subtyping and the notion of power type". In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79, ACM Press, 1988. 3.8.3

[Caromel 89] D. Caromel, "Service, Asynchrony, and Wait-by-Necessity", *Journal of Object-Oriented Programming*, 2(4):12–18, 1989. 5.6.3

[Caromel 93] D. Caromel, "Toward a method of object-oriented concurrent programming", *Communications of the ACM*, 36(9):90–102, 1993. 5.4.2, 5.6.3

[Chambers 04] C. Chambers and T. C. Group, *The Cecil Language: Specification & Rationale*. Technical Report, Department of Computer Science and Engineering, University of Washington, Feb 2004. 3.5

[Coffman 71] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks", *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971. 4.2.2, 4.2.2

[Conway 63] M. E. Conway, "A multiprocessor system design". In *Conference Proceedings 1963 FJCC*, pages 139–146, AFIPS Press, 1963. 4.4.2

[Cook 90] W. R. Cook, W. Hill, and P. S. Canning, "Inheritance is not subtyping". In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, ACM Press, 1990. 3.8.3, 3.8.4

[Courtois 71] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"", *Communications of the ACM*, 14(10):667–668, 1971. 5.10.4

[Dahl 68] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "Some features of the simula 67 language". In *Proceedings of the second conference on Applications of simulations*, pages 29–31, 1968. 3.5, 3.19

[Dennis 66] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations", *Commun. ACM*, 9(3):143–155, 1966. 4.4.2

[Dijkstra 68a] E. W. Dijkstra, *Cooperating Sequential Processes. Programming Languages*, Academic Press, New York, 1968. 4.2.1, 4.2.2, 4.2.2, E.4

[Dijkstra 68b] E. W. Dijkstra, "Cooperating sequential processes". 1968. published as [Dijkstra 68a]. 4.4.1

[Dijkstra 68c] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful", *Communications of the ACM*, 11(3):147–148, 1968. 3.2

[Dijkstra 72] E. W. Dijkstra, "Notes on structured programming". In O.-J. Dahl, E. W. Dijkstra, and C. Hoare, eds., *Structured Programming*, pages 1–82, Academic Press, London and New York, 1972. 2.1.2, 5, 6, 3.2

[ECMA-367 05] "Eiffel analysis, design and programming language". Jun 2005. ECMA-367 Standard. 24, 3.17, A.6

[Floyd 67] R. W. Floyd, "Assigning meanings to programs". In J. T. Schwartz, ed., *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, American Mathematical Society, Providence, 1967. 3.2

[Forum 94] M. P. I. Forum, "MPI: A message-passing interface standard", *International Journal of Supercomputer Applications*, 8(UT-CS-94-230):165–414, 1994. 4.1.3

[Geist 94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge Massachusetts, 1994. 4.1.3

[Ghezzi 91] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice-Hall, 1991. 2.1

[Goldberg 89] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Addison-Wesley, 1989. 3.5, 3.19

[Gosling 05] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. Addison-Wesley, third edition, 2005. 3.5, 3.19, 5.10.3

[Gosling 96] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, first edition, 1996. 3.19, 5.10.3

[Gries 81] D. Gries, *The Science of Programming. Texts and Monographs in Computer Science*, Springer-Verlag, 1981. 10

[Guttag 77] J. Guttag, "Abstract data types and the development of data structures", *Commun. ACM*, 20(6):396–404, 1977. 3.9

[Habermann 69] A. N. Habermann, "Prevention of system deadlocks", *Communications of the ACM*, 12(7):373–377, 1969. 4.2.2

[Harris 03] T. Harris and K. Fraser, "Language support for lightweight transactions". In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, ACM Press, 2003. 5.10.6

[Herlihy 03] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures". In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, ACM Press, 2003. 5.10.6

[Herlihy 87] M. P. Herlihy and J. M. Wing, "Axioms for concurrent objects". In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26, ACM Press, 1987. 5.3.1

[Herlihy 90a] M. Herlihy, "A methodology for implementing highly concurrent data structures". In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, ACM Press, 1990. 5.10.6

[Herlihy 90b] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects", *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. 5.3.1

[Herlihy 91] M. Herlihy, "Wait-free synchronization", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. 5.10.6, 5.10.6

[Herlihy 93] M. Herlihy, "A methodology for implementing highly concurrent data objects", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993. 5.10.6, 5.10.6

[Hoare 69] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, 12(10):576–580, 1969. 3.2

[Hoare 73] C. A. R. Hoare, *Hints on Programming Language Design*. Technical Report STAN-CS-73-403, Stanford Artificial Intelligence Laboratory, Computer Science Department, Stanford University, 1973. 2.1.10, 2.2

[Hoare 74] C. A. R. Hoare, "Monitors: an operating system structuring concept", *Communications of the ACM*, 17(10):549–557, 1974. 5.10.3, 5.11.1, 16, 17

[Hoare 78] C. A. R. Hoare, "Communicating sequential processes", *Communications of the ACM*, 21(8):666–677, 1978. 4.5.2

[Holmes 97] D. Holmes, J. Noble, and J. Potter, "Aspects of synchronization". In *TOOLS '97: Proceedings of the Technology of Object-Oriented Languages and Systems - Tools-25*, page 2, IEEE Computer Society, Washington, DC, USA, 1997. 4.6.1

[Holmes 98] D. Holmes, J. Noble, and J. Potter, "Toward reusable synchronisation for object-oriented languages". In *ECOOP '98: Workshop ion on Object-Oriented Technology*, page 439, Springer-Verlag, London, UK, 1998. 5.9

[Holmes 99] D. Holmes, *Synchronization Rings – Composable Synchronization for Object-Oriented Systems*. PhD thesis, Macquarie University, Sydney, Sydney, Australia, 1999. 15, 5.16

[Issarny 01] V. Issarny, "Concurrent exception handling", *Lecture Notes in Computer Science*, 111–127, 2001. 5.18

[Joung 00] Y.-J. Joung, "Asynchronous group mutual exclusion", *Distributed Computing*, 13(4):189–206, 2000. 5.10.8

[Kafura 89] D. G. Kafura and K. H. Lee, "Inheritance in actor based concurrent object-oriented languages". In *Proceedings of the Third European Conference on Object-Oriented Programming*, July 1989. 5.16

[Knuth 74] D. E. Knuth, "Structured programming with go to statements", *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974. 3.2

[Lamport 77] L. Lamport, "Concurrent reading and writing", *Communications of the ACM*, 20(11):806–811, 1977. 5.10.5

[Lamport 79] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Transactions on Computers*, C-28(9):690–691, 1979. 5.3

[Lamport 83] L. Lamport, "Specifying concurrent program modules", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983. 4.2, 4.2.2

[Lauer 78] H. C. Lauer and R. M. Needham, "On the duality of operating system structures". In *Proceedings of the Second International Symposium on Operating Systems*, October 1978. reprinted in Operating Systems Review, Vol. 13, No. 2, April 1979, pp. 3-19. 4.5, 4.5.4, 5.5

[Lea 00] D. Lea, *Concurrent Programming in Java*. Addison-Wesley, second edition, 2000. 4.6.3, 4.6.4, 8, 5.10.3, 5.11.1, 6.1

[Lieberman 86] H. Lieberman, "Using prototypical objects to implement shared behavior in object-oriented systems". In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, ACM Press, New York, NY, USA, 1986. 3.5

[Liskov 74] B. Liskov and S. Zilles, "Programming with abstract data types". In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974. 3.9

[Liskov 77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in clu", *Communications of the ACM*, 20(8):564–576, 1977. 3.8.1

[Liskov 86] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. MIT Press, Cambridge Massachusetts, 1986. 3.12

[Lu 01] J. Lu, M. Zhang, M. Xu, and D. Yang, "A two-layered-class approach for the reuse of synchronization code.", *Information & Software Technology*, 43(5):287–294, 2001. 5.16

[Madsen 93] O. L. Madsen, B. Moller-Pedersen, and K. Nygaard, *ObjectOriented Programming in the Beta Programming Language*. Addison-Wesley, Jun 1993. 3.5

[Matsuoka 93] S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages". In G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150, MIT Press, 1993. 5.16

[McHale 94] C. McHale, *Synchronization in Concurrent, Object-Oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, University of Dublin, Trinity College, Dublin, Ireland, 1994. 5.16

[Meyer 05] B. Meyer, "Attached types and their application to three open problems of object-oriented programming". In *ECOOP 2005, Proceedings of European Conference on Object-Oriented Programming*, pages 1–32, Springer Verlag, July 2005. A.6

[Meyer 86] B. Meyer, "Genericity versus inheritance". In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 391–405, ACM Press, New York, NY, USA, 1986. 3.10.1

[Meyer 88a] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, N.J., 1988. 2.1, 2.1.11

[Meyer 88b] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, N.J., first edition, 1988. 3.9, 3.19

[Meyer 92] B. Meyer, *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, N.J., March 1992. 2nd printing. 2.2, 2.2.7, 3.5, 3.19, 6.1, 6.3, 6.3.3, 6

[Meyer 97] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997. (document), 3.7, 3.7.1, 3.8.3, 3.9, 3.12, 33, 3.1, 35, 3.12.3, 3.13, 37, 3.19, 3.19, 2, 5.1, 5.2, 5.3, 5.4.4, 5.6.3, 5.14, 5.17.1, 5.18.3, A, A.3, A.4, A.6

[Mitchell 01] S. E. Mitchell, A. Burns, and A. J. Wellings, "Mopping up exceptions", *ACM SIGAda Ada Letters*, XXI(3):80–92, 2001. 5.18

[Moessenboeck 93] H. Moessenboeck, "Object-oriented programming in oberon". 1993. 3.11

[Moore 65] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics*, 38(8), April 1965. 1

[NC 87] H. Norman C, R. K. Raj, A. P. Black, H. M. Levy, and E. Jul, *The Emerald Programming Language*. Technical Report 87-10-07, Department of Computer Science, University of British Columbia, Seattle, WA (USA), 1987. 3.8.2

[Nienaltowski 06a] P. Nienaltowski, "Flexible locking in scoop". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 71–90, University of York – Department of Computer Science, July 2006. 5.12.1, A.6

[Nienaltowski 06b] P. Nienaltowski and B. Meyer, "Contracts for concurrency". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 27–49, University of York – Department of Computer Science, July 2006. 5.14

[OeS 04] M. Oliveira e Silva, "Concurrent object-oriented programming: The MP-Eiffel approach", *Journal of Object Technology: Special issue: TOOLS USA 2003*, 3(4):97–124, April 2004. 5.10.4, 5.18, 5.18.1

[OeS 06a] M. Oliveira e Silva, "Automatic realizations of statically safe intra-object synchronization schemes in MP-Eiffel". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 91–118, University of York – Department of Computer Science, July 2006. Available at `http://www.ieeta.pt/~mos/pubs`. 5.10.4, 5.11.1, 5.11.1, 5.14

[OeS 06b] M. Oliveira e Silva, "Concurrent contracts and inter-object synchronization in MP-Eiffel". 2006. Draft version available at `http://www.ieeta.pt/~mos/pubs`. 23

[Parnas 72a] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, 15(12):1053–1058, 1972. 3.6

[Parnas 72b] D. L. Parnas, "A technique for software module specification with examples", *Communications of the ACM*, 15(5):330–336, 1972. 3.6

[Peterson 83] G. L. Peterson, "Concurrent reading while writing", *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983. 5.10.5, 5.10.5

[Pierce 02] B. C. Pierce, *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002. 2.2.4, 3.1, 3.8.2

[Puntigam 05] F. Puntigam, "Client and server synchronization expressed in types". In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005. 4.6.1

[Ruschitzka 77] M. Ruschitzka and R. S. Fabry, "A unifying approach to scheduling", *Communications of the ACM*, 20(7):469–477, 1977. 4.1.4

[Ryant 97] I. Ryant, "Why inheritance means extra trouble", *Communications of the ACM*, 40(10):118–119, 1997. 3.11.3

[Strachey 00] C. Strachey, "Fundamental concepts in programming languages", *Higher Order Symbol. Comput.*, 13(1-2):11–49, 2000. (reprinted from 1967 article). 3.10

[Stroustrup 85] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, first edition, 1985. 3.19

[Stroustrup 97] B. Stroustrup, *The C++ Programming Language*. Addison Wesley Longman, third edition, 1997. 3.5, 3.19

[Sun Microsystems Java Specification Requests 04] Sun Microsystems, Java Specification Requests, "JSR166: Concurrency Utilities". 2004. (`http://www.jcp.org/en/jsr/detail?id=166`). 5.10.6

[Templ 93] J. Templ, "A systematic approach to multiple inheritance implementation", *SIGPLAN Not.*, 28(4):61–66, 1993. 3.11

[Ungar 87] D. Ungar and R. B. Smith, "Self: The power of simplicity". In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, ACM Press, New York, NY, USA, 1987. 3.5

[Ungar 91] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle, "Organizing programs without classes", *Lisp and Symbolic Computation*, 4(3), June 1991. 3.5

[Wirth 71] N. Wirth, "Program development by stepwise refinement", *Communications of the ACM*, 14(4):221–227, 1971. 3.2

[Wirth 74] N. Wirth, "On the composition of well-structured programs", *ACM Computing Surveys (CSUR)*, 6(4):247–259, 1974. 3.2

[Wirth 85] N. Wirth, *Programming in MODULA-2 (3rd corrected ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1985. 3.8.1

[Xu 95] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery". In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995. 5.18